(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 917 056 A2

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
19.05.1999 Bulletin 1999/20

(51) Int Cl.[6]: G06F 9/46

(21) Application number: 98309006.9

(22) Date of filing: 04.11.1998

(84) Designated Contracting States:
AT BE CH CY DE DK ES FI FR GB GR IE IT LI LU MC NL PT SE
Designated Extension States:
AL LT LV MK RO SI

(30) Priority: 04.11.1997 US 64250 P
10.06.1998 US 95368
10.06.1998 US 95265
10.06.1998 US 95521
10.06.1998 US 95188
10.06.1998 US 90027
10.06.1998 US 95543
10.06.1998 US 95379
10.06.1998 US 95277
10.06.1998 US 95266
10.06.1998 US 95264
10.06.1998 US 95256

(71) Applicant: DIGITAL EQUIPMENT CORPORATION
Houston, Texas 77070-2698 (US)

(72) Inventors:
- Noel, Karen
Pembroke, NH 03275 (US)
- Benson, Thomas
Hollis, NH 03049 (US)
- Jordan, Gregory H.
Hollis, NH 03049 (US)
- Kauffman, James R.
Nashua, NH 03062 (US)
- Mason, Andrew H.
Hollis, NH 03049 (US)
- Harter, Paul K.
Groton, MA 01450 (US)
- Bishop, Richard A.
Merrimack, NH 03054 (US)
- Kleinsorge, Frederick G.
Amherst, NH 03031 (US)
- Shirron, Stephen
Acton, MA 01720 (US)
- Zalewski, Stephen
Redmond, WA 98052 (US)

(74) Representative: Brunner, Michael John
GILL JENNINGS & EVERY
Broadgate House
7 Eldon Street
London EC2M 7LH (GB)

(54) A multi-processor computer system and a method of operating thereof

(57) A computer system has a plurality of assignable system resources, including processors, memory and I/O circuitry; an interconnection mechanism for electrically interconnecting the processors, memory and I/O circuitry so that each processor has electrical access to all the memory and at least some of the I/O circuitry; a software mechanism for assigning the assignable system resources to a plurality of partitions, each partition including at least one processor, some memory and some I/O circuitry; and an operating system instance running in each partition. The computer system provides improved flexibility, resource availability, resource migration capabilities and scalability.

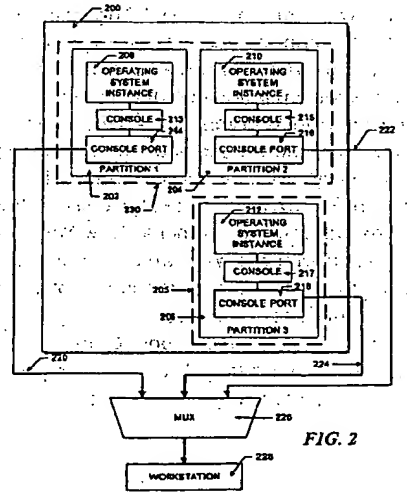


FIG. 2

EP 0 917 056 A2

BEST AVAILABLE COPY

Printed by Jouve, 75001 PARIS (FR)

[0008] The VM architecture supports the concept of a "logical partition" or LPAR. Each LPAR contains some of the available physical CPUs and resources which are logically assigned to the partition. The same resources can be assigned to more than one partition. LPARs are set up by an administrator statically, but can respond to changes in load dynamically, and without rebooting, in several ways. For example, if two logical partitions, each containing ten CPUs, are shared on a physical system containing ten physical CPUs, and, if the logical ten CPU partitions have complementary peak loads, each partition can take over the entire physical ten CPU system as the workload shifts without a reboot or operator intervention.

[0009] In addition, the CPUs logically assigned to each partition can be turned "on" and "off" dynamically via normal operating system operator commands without re-boot. The only limitation is that the number of CPUs active at system initialization is the maximum number of CPUs that can be turned "on" in any partition.

[0010] Finally, in cases where the aggregate workload demand of all partitions is more than can be delivered by the physical system, LPAR "weights" can be used to define the portion of the total CPU resources which is given to each partition. These weights can be changed by system administrators, on-the-fly, with no disruption.

[0011] Another prior art system is called a "Parallel Sysplex" and is also marketed and developed by the International Business Machines Corporation. This architecture consists of a set of computers that are clustered via a hardware entity called a "coupling facility" attached to each CPU. The coupling facilities on each node are connected via a fiber-optic link, and each node operates as a traditional SMP machine, with a maximum of 10 CPUs. Certain CPU instructions directly invoke the coupling facility. For example, a node registers a data structure with the coupling facility, then the coupling facility takes care of keeping the data structures coherent within the local memory of each node.

[0012] The Enterprise 10000 Unix server developed and marketed by Sun Microsystems, Mountain View, California, uses a partitioning arrangement called "Dynamic System Domains" to logically divide the resources of a single physical server into multiple partitions, or domains, each of which operates as a stand-alone server. Each of the partitions has CPUs, memory and I/O hardware. Dynamic reconfiguration allows a system administrator to create, resize, or delete domains "on the fly" and without rebooting. Every domain remains logically isolated from any other domain in the system, isolating it completely from any software error or CPU, memory, or I/O error generated by any other domain. There is no sharing of resources between any of the domains.

[0013] The Hive Project conducted at Stanford University uses an architecture which is structured as a set of cells. When the system boots, each cell is assigned a range of nodes, each having memory and I/O devices, that the cell owns throughout execution. Each cell manages the processors, memory and I/O devices on those nodes as if it were an independent operating system. The cells cooperate to present the illusion of a single system to user-level processes.

[0014] Hive cells are not responsible for deciding how to divide their resources between local and remote requests. Each cell is responsible only for maintaining its internal resources and for optimizing performance within the resources it has been allocated. Global resource allocation is carried out by a user-level process called "wax." The Hive system attempts to prevent data corruption by using certain fault containment boundaries between the cells. In order to implement the tight sharing expected from a multiprocessor system, despite the fault containment boundaries between cells, resource sharing is implemented through the cooperation of the various cell kernels, but the policy is implemented outside the kernels in the wax process. Both memory and processors can be shared.

[0015] A system called "Cellular IRIX" developed and marketed by Silicon Graphics Inc. Mountain View, California, supports modular computing by extending traditional symmetric multiprocessing systems. The Cellular IRIX architecture distributes global kernel text and data into optimized SMP-sized chunks or "cells". Cells represent a control domain consisting of one or more machine modules, where each module consists of processors, memory, and I/O. Applications running on these cells rely extensively on a full set of local operating system services, including local copies of operating system text and kernel data structures, bit only one instance of the operating system exists on the entire system. Inter-cell coordination allows application images to directly and transparently utilize processing, memory and I/O resources from other cells without incurring the overhead of data copies or extra context switches.

[0016] Another existing architecture called NUMA-Q developed and marketed by Sequent Computer Systems, Inc., Beaverton, Oregon uses "quads", or a group of four processors per portion of memory, as the basic building block for NUMA-Q SMP nodes. Adding I/O to each quad further improves performance. Therefore, the NUMA-Q architecture not only distributes physical memory but puts a predetermined number of processors and PCI slots next to each processor. The memory in each quad is not local memory in the traditional sense. Rather, it is a portion of the physical memory address space and has a specific address range. The address map is divided evenly over memory, with each quad containing a contiguous portion of address space. Only one copy of the operating system is running and, as in any SMP system, it resides in memory and runs processes without distinction and simultaneously on one or more processors.

[0017] Accordingly, while many attempts have been made at providing a flexible computer system, existing systems each have significant shortcomings. Therefore, it would be desirable to have a new computer system design which provides improved flexibility, resource availability, resource migration capabilities and scalability.

[0018] The invention resides in a computer system having a plurality of assignable system resources, including

a failure has occurred in the local instance.

[0046] Figures 26A-26B, when placed together, form a flowchart that shows the general procedure followed for resource relocation after a failure occurs in an instance of a first partition.

[0047] Figure 27 is a flowchart showing the procedure followed by a CPU migrating from an instance within which a failure has occurred.

[0048] Figure 28 is a flowchart illustrating the steps followed during initial assignment of permanent ownership of a CPU of the present invention.

[0049] Figure 29 is a flowchart illustrating the steps followed during a change of permanent ownership of a CPU of the present invention.

[0050] Figures 30A-30C, when placed together, form a flowchart illustrating the steps in an illustrative routine followed by the system when a CPU is migrating under a "HALT" type migration.

[0051] Figure 31 is a flowchart illustrating the steps in an illustrative routine followed by the system when a CPU in a halted state is migrated.

[0052] Figures 32A-32B, when placed together, form a flowchart illustrating the steps in an illustrative routine followed by the system when a CPU is migrated by deassigning it and subsequently reassigning it. Figures 33A and 33B, when placed together, form a flowchart illustrating the steps in a routine followed by two operating system instances to communicate from one instance to the other using single-bit notification.

[0053] Figures 34A and 34B, when placed together, form a flowchart illustrating the steps in a routine followed by two operating system instances to communicate from one instance to the other using packetized data transfer.

A. OVERALL SYSTEM

[0054] A computer platform constructed in accordance with the principles of the present invention is a multi-processor system capable of being partitioned to allow the concurrent execution of multiple instances of operating system software. The system does not require hardware support for the partitioning of its memory, CPUs and I/O subsystems, but some hardware may be used to provide additional hardware assistance for isolating faults, and minimizing the cost of software engineering. The following specification describes the interfaces and data structures required to support the inventive software architecture. The interfaces and data structures described are not meant to imply a specific operating system must be used, or that only a single type of operating system will execute concurrently. Any operating system which implements the software requirements discussed below can participate in the inventive system operation.

[0055] System Building BlocksThe inventive software architecture operates on a hardware platform which incorporates multiple CPUs, memory and I/O hardware. Preferably, a modular architecture such as that shown in Figure 1 is used, although those skilled in the art will understand that other architectures can also be used, which architectures need not be modular. Figure 1 illustrates a computing system constructed of four basic system building blocks (SBBs) 100 - 106. In the illustrative embodiment, each building block, such as block 100, is identical and comprises several CPUs 108 - 114, several memory slots (illustrated collectively as memory 120), an I/O processor 118, and a port 116 which contains a switch (not shown) that can connect the system to another such system. However, in other embodiments, the building blocks need not be identical. Large multiprocessor systems can be constructed by connecting the desired number of system building blocks by means of their ports. Switch technology, rather than bus technology, is employed to connect building block components in order to both achieve the improved bandwidth and to allow for non-uniform memory architectures (NUMA).

[0056] In accordance with the principles of the invention, the hardware switches are arranged so that each CPU can address all available memory and I/O ports regardless of the number of building blocks configured as schematically illustrated by line 122. In addition, all CPUs may communicate to any or all other CPUs in all SBBs with conventional mechanisms, such as inter-processor interrupts. Consequently, the CPUs and other hardware resources can be associated solely with software. Such a platform architecture is inherently scalable so that large amounts of processing power, memory and I/O will be available in a single computer.

[0057] An APMP computer system 200 constructed in accordance with the principles of the present invention from a software view is illustrated in Figure 2. In this system, the hardware components have been allocated to allow concurrent execution of multiple operating system instances 208, 210, 212.In a preferred embodiment, this allocation is performed by a software program called a "console" program, which, as will hereinafter be described in detail, is loaded into memory at power up. Console programs are shown schematically in Figure 2 as programs 213, 215 and 217. The console program may be a modification of an existing administrative program or a separate program which interacts with an operating system to control the operation of the preferred embodiment. The console program does not virtuallze the system resources, that is, it does not create any software layers between the running operating systems 208, 210 and 212 and the physical hardware, such as memory and I/O units (not shown in Figure 2.) Nor is the state of the running operating systems 208, 210 and 212 swapped to provide access to the same hardware. Instead, the inventive system logically divides the hardware into partitions. It is the responsibility of operating system instance 208, 210, and

creates and initializes the partitions based on the environment variables. In this latter process the master console operates to create the configuration tree, to create additional HWRPB data blocks, to load the additional console program copies, and to start the CPUs on the alternate HWRPBs. Each partition then has an operating system instance running on it, which instance cooperates with a console program copy also running in that partition. In an unconfigured APMP system, the master console program will initially create a single partition containing the primary CPU, a minimum amount of memory, and a physical system administrator's console selected in a platform-specific way. Console program commands will then allow the system administrator to create additional partitions, and configure I/O buses, memory, and CPUs for each partition.

[0065] After associations of resources to partitions have been made by the console program, the associations are stored in non-volatile RAM to allow for an automatic configuration of the system during subsequent boots. During subsequent boots, the master console program must validate the current configuration with the stored configuration to handle the removal and addition of new components. Newly-added components are placed into an unassigned state, until they are assigned by the system administrator. If the removal of a hardware component results in a partition with insufficient resources to run an operating system, resources will continue to be assigned to the partition, but it will be incapable of running an operating system instance until additional new resources are allocated to it. As previously mentioned, the console program communicates with an operating system instance by means of an HWRPB which is passed to the operating system during operating system boot up. The fundamental requirements for a console program are that it should be able to create multiple copies of HWRPBs and itself. Each HWRPB copy created by the console program will be capable of booting an independent operating system instance into a private section of memory and each operating system instance booted in this manner can be identified by a unique value placed into the HWRPB. The value indicates the partition, and is also used as the operating system instance ID.

[0066] In addition, the console program is configured to provide a mechanism to remove a CPU from the available CPUs within a partition in response to a request by an operating system running in that partition. Each operating system instance must be able to shutdown, halt, or otherwise crash in a manner that control is passed to the console program. Conversely, each operating system instance must be able to reboot into an operational mode, independently of any other operating system instance.

[0067] Each HWRPB which is created by a console program will contain a CPU slot-specific database for each CPU that is in the system, or that can be added to the system without powering the entire system down. Each CPU that is physically present will be marked "present", but only CPUs that will initially execute in a specific partition will be marked "available" in the HWRPB for the partition. The operating system instance running on a partition will be capable of recognizing that a CPU may be available at some future time by a present (PP) bit in a per-CPU state flag fields of the HWRPB, and can build data structures to reflect this. When set, the available (PA) bit in the per-CPU state flag fields indicates that the associated CPU is currently associated with the partition, and can be invited to join SMP operation.

The Configuration Tree

[0068] As previously mentioned, the master console program creates a configuration tree which represents the hardware configuration, and the assignment of each component in the system to each partition. Each console program then identifies the configuration tree to its associated operating system instance by placing a pointer to the tree in the HWRPB. Referring to Figure 3, the configuration tree 300 represents the hardware components in the system, the platform constraints and minimums, and the software configuration. The master console program builds the tree using information discovered by probing the hardware, and from information stored in non-volatile RAM which contains configuration information generated during previous initializations.

[0069] The master console may generate a single copy of the tree which copy is shared by all operating system instances, or it may replicate the tree for each instance. A single copy of the tree has the disadvantage that it can create a single point of failure in systems with independent memories. However, platforms that generate multiple tree copies require the console programs to be capable of keeping changes to the tree synchronized.

[0070] The configuration tree comprises multiple nodes including root nodes, child nodes and sibling nodes. Each node is formed of a fixed header and a variable length extension for overlaid data structures. The tree starts with a tree root node 302 representing the entire system box, followed by branches that describe the hardware configuration (hardware root node 304), the software configuration (software root node 306), and the minimum partition requirements (template root node 308.) In Figure 3, the arrows represent child and sibling relationships. The children of a node represent component parts of the hardware or software configuration. Siblings represent peers of a component that may not be related except by having the same parent. Nodes in the tree 300 contain information on the software communities and operating system instances, hardware configuration, configuration constraints, performance boundaries and hot-swap capabilities. The nodes also provide the relationship of hardware to software ownership, or the sharing of a hardware component. The nodes are stored contiguously in memory and the address offset from the tree root node 302 of the tree 300 to a specific node forms a "handle" which may be used from any operating system

or are configurable only as a part of another component (a device adapter, for example, may be configurable only as a part of its bus). A partition which is, as explained above, a grouping of CPUs, memory, and I/O devices into a unique software entity also has minimum requirements. For example, the minimum hardware requirements for a partition are at least one CPU, some private memory (platform dependent minimum, including console memory) and an I/O bus, including a physical, non-shared, console port.

[0078] The minimal component requirements for a partition are provided by the information contained in the template root node 308. The template root node 308 contains nodes, 316, 318 and 320, representing the hardware components that must be provided to create a partition capable of execution of a console program and an operating system instance. Configuration editors can use this information as the basis to determine what types, and how many resources must be available to form a new partition.

[0079] During the construction of a new partition, the template subtree will be "walked", and, for each node in the template subtree, there must be a node with the same type and subtype owned by the new partition so that it will be capable of loading a console program and booting an operating system instance. If there are more than one node of the same type and subtype in the template tree, there must also be multiple nodes in the new partition. The console program will use the template to validate that a new partition has the minimum requirements prior to attempting to load a console program and initialize operation.

[0080] The following is a detailed example of a particular implementation of configuration tree nodes. It is intended for descriptive purposes only and is not intended to be limiting. Each HWRPB must point to a configuration tree which provides the current configuration, and the assignments of components to partitions. A configuration pointer (in the CONFIG field) in the HWRPB is used to point to the configuration tree. The CONFIG field points to a 64-byte header containing the size of the memory pool for the tree, and the initial checksum of the memory. Immediately following the header is the root node of the tree. The header and root node of the tree will be page aligned.

[0081] The total size in bytes of the memory allocated for the configuration tree is located in the first quadword of the header. The size is guaranteed to be in multiples of the hardware page size. The second quadword of the header is reserved for a checksum. In order to examine the configuration tree, an operating system instance maps the tree into its local address space. Because an operating system instance may map this memory with read access allowed for all applications, some provision must be made to prevent a non-privileged application from gaining access to console data to which it should not have access. Access may be restricted by appropriately allocating memory. For example, the memory may be page aligned and allocated in whole pages. Normally, an operating system instance will map the first page of the configuration tree, obtain the tree size, and then remap the memory allocated for configuration tree usage. The total size may include additional memory used by the console for dynamic changes to the tree.

[0082] Preferably, configuration tree nodes are formed with fixed headers, and may optionally contain type-specific information following the fixed portion. The size field contains the full length of the node, nodes are illustratively allocated in multiples of 64-bytes and padded as needed. The following description defines illustrative fields in the fixed header for a node:

```
typedef struct _gct_node {
        unsigned char       type;
        unsigned char       subtype;
```

[0083] In the above definition the type definitions "uint" are unsigned integers with the appropriate bit lengths. As previously mentioned, nodes are located and identified by a handle (identified by the typedef GCT_HANDLE in the definition above). An illustrative handle is a signed 32-bit offset from the base of the configuration tree to the node. The value is unique across all partitions in the computer system. That is, a handle obtained on one partition must be valid to lookup a node, or as an input to a console callback, on all partitions. The magic field contains a predetermined bit pattern which indicates that the node is actually a valid node.

[0084] The tree root node represents the entire system. Its handle is always zero. That is, it is always located at the first physical location in the memory allocated for the configuration tree following the config header. It has the following definition:

```
typedef struct _gct_root_node {
        GCT_NODE        hd;
        uint64          lock;
        uint64          transient_level;
        uint64          current_level;
        uint64          console_req;
        uint64          min_alloc;
        uint64          min_align;
        uint64          base_alloc;
        uint64          base_align;
        uint64          max_phys_address;
        uint64          mem_size;
        uint64          platform_type;
        int32           platform_name;
        GCT_HANDLE      primary_instance;
        GCT_HANDLE      first_free;
        GCT_HANDLE      high_limit;
        GCT_HANDLE      lookaside;
        GCT_HANDLE      available;
```

high_limit

This field holds the highest address at which a valid node can be located within the configuration tree. It is used by callbacks to validate that a handle is legal.

lookaside

This field is the handle of a linked list of nodes that have been deleted, and that may be reclaimed. When a community or partition are deleted, the node is linked into this list, and creation of a new partition or community will look at this list before allocating from free pool.

available

This field holds the number of bytes remaining in the free pool pointed to by the first_free field.

max_partitions

This field holds the maximum number of partitions computed by the platform based on the amount of hardware resources currently available.

partitions

This field holds an offset from the base of the root node to an array of handles. Each partition ID is used as an index into this array, and the partition node handle is stored at the indexed location. When a new partition is created, this array is examined to find the first partition ID which does not have a corresponding partition node handle and this partition ID is used as the ID for the new partition.

communities

This field also holds an offset from the base of the root node to an array of handles. Each community ID is used an index into this array, and a community node handle is stored in the array. When a new community is created, this array is examined to find the first community ID which does not have a corresponding community node handle and this community ID is used as the ID for the new community. There cannot be more communities than partitions, so the array is sized based on the maximum number of partitions.

max_platform_partition

This field holds the maximum number of partitions that can simultaneously exist on the platform, even if additional hardware is added (potentially inswapped).

max_fragments

This field holds a platform defined maximum number of fragments into which a memory descriptor can be divided. It is used to size the array of fragments in the memory descriptor node.

max_desc

This field holds the maximum number of memory descriptors for the platform.

APMP_id

This field holds a system ID set by system software and saved in non-volatile RAM.

APMP_id_pad

This field holds padding bytes for the APMP ID.

bindings

This field holds an offset to an array of "bindings" Each binding entry describes a type of hardware node, the type of node the parent must be, the configuration binding, and the affinity binding for a node type. Bindings are used by software to determine how node types are related and configuration and affinity rules.

[0086] A community provides the basis for the sharing of resources between partitions. While a hardware component may be assigned to any partition in a community, the actual sharing of a device, such as memory, occurs only within a community. The community node 310 contains a pointer to a control section, called an APMP database, which allows the operating system instances to control access and membership in the community for the purpose of sharing memory and communications between instances. The APMP database and the creation of communities are discussed in detail below. The configuration ID for the community is a signed 16-bit integer value assigned by the console program. The ID value will never be greater than the maximum number of partitions that can be created on the platform.

[0087] A partition node, such as node 312 or 314, represents a collection of hardware that is capable of running an independent copy of the console program, and an independent copy of an operating system. The configuration ID for this node is a signed 16-bit integer value assigned by the console. The ID will never be greater than the maximum number of partitions that can be created on the platform. The node has the definition:

```
typedef struct_gct_cpu_node {
        GCT_NODE    hd;
} GCT_CPU_NODE;
```

[0091] A memory subsystem node, such as node 334 or 348, is a "pseudo" node that groups together nodes representing the physical memory controllers and the assignments of the memory that the controllers provide. The children of this node consist of one or more memory controller nodes (such as nodes 336 and 350) which the console has configured to operate together (interleaved), and one or more memory descriptor nodes (such as nodes 338 and 352) which describe physically contiguous ranges of memory.

[0092] A memory controller node (such as nodes 336 or 350) is used to express a physical hardware component, and its owner is typically the partition which will handle errors, and initialization. Memory controllers cannot be assigned to communities, as they require a specific operating system instance for initialization, testing and errors. However, a memory description, defined by a memory descriptor node, may be split into "fragments" to allow different partitions or communities to own specific memory ranges within the memory descriptor. Memory is unlike other hardware resources in that it may be shared concurrently, or broken into "private" areas. Each memory descriptor node contains a list of subset ranges that allow the memory to be divided among partitions, as well as shared between partitions (owned by a community). A memory descriptor node (such as nodes 338 or 352) is defined as :

```
typedef struct_gct_mem_desc_node {
    GCT_NODE      hd;
    GCT_MEM_INFO  mem_info;
    int32         mem_frag;
}GCT_MEM_DESC_NODE;
```

[0093] The mem_info structure has the following definition:

```
typedef struct_gct_mem_info {
    uint64    base_pa;
    uint64    base_size;
    uint32    desc_count;
    uint32    info_fill;
}GCT_MEM_INFO:
```

[0094] The mem_frag field holds an offset from the base of the memory descriptor node to an array of GCT_MEM_DESC structures which have the definition:.

The configuration tree has ownership pointers illustrated in Figure 4 which determine the mapping of hardware devices to software such as partitions (exclusive access) and communities (shared access). An operating system instance uses the information in the configuration tree to determine to which hardware resources it has access and reconfiguration control.

[0099] Passive hardware resources which have no owner are unavailable for use until ownership is established. Once ownership is established by altering the configuration tree, the operating system instances may begin using the resources. When an instance makes an initial request, ownership can be changed by causing the owning operating system to stop using a resource or by a console program taking action to stop using a resource in a partition where no operating system instance is executing. The configuration tree is then altered to transfer ownership of the resource to another operating system instance. The action required to cause an operating system to stop using a hardware resource is operating system specific, and may require a reboot of the operating system instances affected by the change.

[0100] To manage the transition of a resource from an owned and active state, to a unowned and inactive state, two fields are provided in each node of the tree. The owner field represents the owner of a resource and is loaded with the handle of the owning software partition or community. At power up of an APMP system, the owner fields of the hardware nodes are loaded from the contents of non-volatile RAM to establish an initial configuration.

[0101] To change the owner of a resource, the handle value is modified in the owner field of the hardware component, and in the owner fields of any descendants of the hardware component which are bound to the component by their config handles. The current_owner field represents the current user of the resource. When the owner and current_owner fields hold the same non-zero value, the resource is owned and active. Only the owner of a resource can de-assign the resource (set the owner field to zero). A resource that has null owner and current_owner fields is unowned, and inactive. Only resources which have null owner and current_owner fields may be assigned to a new partition or community.

[0102] When a resource is de-assigned, the owner may decide to de-assign the owner field, or both the owner and current_owner fields. The decision is based on the ability of the owning operating system instance running in the partition to discontinue the use of the resource prior to de-assigning ownership. In the case where a reboot is required to relinquish ownership, the owner field is cleared, but the current_owner field is not changed. When the owning operating system instance reboots, the console program can clear any current_owner fields for resources that have no owner during initialization.

[0103] During initialization, the console program will modify the current_owner field to match the owner field for any node of which it is the owner, and for which the current_owner field is null. System software should only use hardware of which it is the current owner. In the case of a de-assignment of a resource which is owned by a community, it is the responsibility of system software to manage the transition between states. In some embodiments, a resource may be loaned to another partition. In this condition, the owner and current_owner fields are both valid, but not equal. The following table summarizes the possible resource states and the values of the owner and current_owner fields:

TABLE 1

Owner field value	current_owner field value	Resource State
None	none	unowned, and inactive
None	valid	unowned, but still active
Valid	none	owned, not yet active
Valid	equal to owner	owned and active
Valid	is not equal to owner	loaned

[0104] Because CPUs are active devices, and sharing of CPUs means that a CPU could be executing in the context of a partition which may not be its "owner", ownership of a CPU is different from ownership of a passive resource. The CPU node in the configuration tree provides two fields that indicate which partition a CPU is nominally "owned" by, and in which partition the CPU is currently executing. The owner field contains a value which indicates the nominal ownership of the CPU, or more specifically, the partition in which the CPU will initially execute at system power up.

[0105] Until an initial ownership is established (that is, if the owner field is unassigned), CPUs are placed into a HWRPB context decided by the master console, but the HWRPB available bit for the CPU will not be set in any HWRPB. This combination prevents the CPU from joining any operating system instance in SMP operation. When ownership of a CPU is established (the owner field is filled in with a valid partition handle), the CPU will migrate, if necessary, to the owning partition, set the available bit in the HWRPB associated with that partition, and request to join SMP operation of the instance running in that partition, or join the console program in SMP mode. The combination of the present an available bits in the HWRPB tell the operating system instance that the CPU is available for use in SMP operation, and

memory" for communicating or sharing data with other instances running within the computer if the instance is part of a system in which memory is shared. In such a shared memory system, an instance may need to be capable of mapping physical "shared memory" as identified in the configuration tree into its virtual address space, and the virtual address spaces of the "processes" running within that operating system instance.

7. Each instance may need some mechanism to contact another CPU in the computer system in order to communicate with it.

8. An instance may also need to be able to recognize other CPUs that are compatible with its operations, even if the CPUs are not currently assigned to its partition. For example, the instance may need to be able to ascertain CPU parameters, such as console revision number and clock speed, to determine whether it could run with that CPU, if the CPU was re-assigned to the partition in which the instance is running.

Changing the Configuration Tree

[0113] Each console program provides a number of callback functions to allow the associated operating system instance to change the configuration of the APMP system, for example, by creating a new community or partition, or altering the ownership of memory fragments. In addition, other callback functions provide the ability to remove a community, or partition, or to start operation on a newly-created partition.

[0114] However, callback functions do not cause any changes to take place on the running operating system instances. Any changes made to the configuration tree must be acted upon by each instance affected by the change. The type of action that must take place in an instance when the configuration tree is altered is a function of the type of change, and the operating system instance capabilities. For example, moving an input/output processor from one partition to another may require both partitions to reboot. Changing the memory allocation of fragments, on the other hand, might be handled by an operating system instance without the need for a reboot.

[0115] Configuration of an APMP system entails the creation of communities and partitions, and the assignment of unassigned components. When a component is moved from one partition to another, the current owner removes itself as owner of the resource and then indicates the new owner of the resource. The new owner can then use the resource. When an instance running in a partition releases a component, the instance must no longer access the component. This simple procedure eliminates the complex synchronization needed to allow blind stealing of a component from an instance, and possible race conditions in booting an instance during a reconfiguration.

[0116] Once initialized, configuration tree nodes will never be deleted or moved, that is, their handles will always be valid. Thus, hardware node addresses may be cached by software. Callback functions which purport to delete a partition or a community do not actually delete the associated node, or remove it from the tree, but instead flag the node as UNAVAILABLE, and clear the ownership fields of any hardware resource that was owned by the software component.

[0117] In order to synchronize changes to the configuration tree, the root node of the tree maintains two counters (transient_level and current_level). The transient_level counter is incremented at the start of an update to the tree, and the current_level counter is incremented when the update is complete. Software may use these counters to determine when a change has occurred, or is occurring to the tree. When an update is completed by a console, an interrupt can be generated to all CPUs in the APMP system. This interrupt can be used to cause system software to update its state based on changes to the tree.

Creation of an APMP Computer System

[0118] Figure 5 is a flowchart that illustrates an overview of the formation of the illustrative adaptively-partitioned, multi-processor (APMP) computer system. The routine starts in step 500 and proceeds to step 502 where a master console program is started. If the APMP computer system is being created on power up, the CPU on which the master console runs is chosen by a predetermined mechanism, such as arbitration, or another hardware mechanism. If the APMP computer system is being created on hardware that is already running, a CPU in the first partition that tries to join the (non-existent) system runs the master console program, as discussed below.

[0119] Next, in step 504, the master console program probes the hardware and creates the configuration tree in step 506 as discussed above. If there is more than one partition in the APMP system on power up, each partition is initialized and its console program is started (step 508).

[0120] Finally, an operating system instance is booted in at least one of the partitions as indicated in step 510. The first operating system instance to boot creates an APMP database and fills in the entries as described below. APMP databases store information relating to the state of active operating system instances in the system. The routine then finishes in step 512. It should be noted that an instance is not required to participate in an APMP system. The instance can choose not to participate or to participate at a time that occurs well after boot. Those instances which do participate form a "sharing set." The first instance which decides to join a

[0121] As previously mentioned, the illustrative computer system can operate with several different operating systems

figuration tree. The purpose of allocating two ranges is to permit failover in case of hardware memory failure. Memory management is responsible for mapping the physical memory into virtual address space for the APMP database.

[0126] The detailed actions taken by an operating system instance are illustrated in Figure 6. More specifically, when an operating system instance wishes to become a member of a sharing set, it must be prepared to create the APMP computer system if it is the first instance attempting to "join" a non-existent system. In order for the instance to determine whether an APMP system already exists, the instance must be able to examine the state of shared memory as described above. Further, it must be able to synchronize with other instances which may be attempting to join the APMP system and the sharing set at the same time to prevent conflicting creation attempts. The master console creates the configuration tree as discussed above. Subsequently, a region of memory is initialized by the first, or primary, operating system instance to boot, and this memory region can be used for an APMP database.

Mapping the APMP Database Header

[0127] The goal of the initial actions taken by all operating system instances is to map the header portion of the APMP database and initialize primitive inter-instance interrupt handling to lay the groundwork for a create or join decision. The routine used is illustrated in Figure 6 which begins in step 600. The first action taken by each instance (step 602) is to engage memory management to map the initial segment of the APMP database as described above. At this time, the array of node blocks in the second database section is also mapped. Memory management maps the initial and second segments of the APMP database into the primary operating system address space and returns the start address and length. The instance then informs the console to store the location and size of the segments in the configuration tree.

[0128] Next, in step 604, the initial virtual address of the APMP database is used to allow the initialization routine to zero interrupt reason masks in the node block assigned to the current instance.

[0129] A zero initial value is then stored to the heartbeat field for the instance in the node block, and other node block fields. In some cases, the instance attempting to create a new APMP computer system was previously a member of an APMP system and did not withdraw from the APMP system. If this instance is rebooting before the other instances have removed it, then its bit will still be "on" in the system membership mask. Other unusual or error cases can also lead to "garbage" being stored in the system membership mask.

[0130] Next, in step 608, the virtual address (VA) of the APMP database is stored in a private cell which is examined by an inter-processor interrupt handler. The handler examines this cell to determine whether to test the per-instance interrupt reason mask in the APMP database header for work to do. If this cell is zero, the APMP database is not mapped and nothing further is done by the handler. As previously discussed, the entire APMP database, including this mask, is initialized so that the handler does nothing before the address is stored. In addition, a clock interrupt handler can examine the same private cell to determine whether to increment the instance-specific heartbeat field for this instance in the appropriate node block. If the private cell is zero, the interrupt handler does not increment the heartbeat field.

[0131] At this point, the routine is finished (step 610) and the APMP database header is accessible and the joining instance is able to examine the header and decide whether the APMP computer system does not exist and, therefore, the instance must create it, or whether the instance will be joining an already-existing APMP system.

[0132] Once the APMP header is mapped, the header is examined to determine whether an APMP computer system is up and functioning, and, if not, whether the current instance should initialize the APMP database and create the APMP computer system. The problem of joining an existing APMP system becomes more difficult, for example, if the APMP computer system was created at one time, but now has no members, or if the APMP system is being reformed after an error. In this case, the state of the APMP database memory is not known in advance, and a simple memory test is not sufficient. An instance that is attempting to join a possibly existing APMP system must be able to determine whether an APMP system exists or not and, if it does not, the instance must be able to create a new APMP system without interference from other instances. This interference could arise from threads running either on the same instance or on another instance.

[0133] In order to prevent such interference, the create/join decision is made by first locking the APMP database and then examining the APMP header to determine whether there is a functioning APMP computer system. If there is a properly functioning APMP system, then the instance joins the system and releases the lock on the APMP database. Alternatively, if there is no APMP system, or if the there is an APMP system, but it is non-functioning, then the instance creates a new APMP system, with itself as a member and releases the lock on the APMP database.

[0134] If there appears to be an APMP system in transition, then the instance waits until the APMP system is again operational or dead, and then proceeds as above. If a system cannot be created, then joining fails.

- little brother" scheme. More particularly, when an instance joins the APMP system, before releasing the lock on the APMP database, it picks one of the current members to be its big brother and watch over the joining instance. The joining instance first assumes big brother duties for its chosen big brother's current little brother, and then assigns itself as the new little brother of the chosen instance. Conversely, when an instance exits the APMP computer system while still in operation so that it is able to perform exit processing, and while it is holding the lock on the APMP database, it assigns its big brother duties to its current big brother before it stops incrementing its heartbeat.

[0142] Every clock tick, after incrementing its own heartbeat, each instance reads its little brother's heartbeat and compares it to the value read at the last clock tick. If the new value is greater, or the little brother's ID has changed, the little brother is considered active. However, if the little brother ID and its heartbeat value are the same, the little brother is considered inactive, and the current instance begins watching its little brother's little brother as well. This accumulation of responsibility continues to a predetermined maximum and insures that the failure of one instance does not result in missing the failure of its little brother. If the little brother begins incrementing its heartbeat again, all additional responsibilities are dropped.

[0143] If a member instance is judged dead, or disinterested, and it has not notified the APMP computer system of its intent to shut down or crash, the instance is removed from the APMP system. This may be done, for example, by setting the "bugcheck" bit in the instance primitive interrupt mask and sending an IP interrupt to all CPU's of the instance. As a rule, shared memory may only be accessed below the hardware priority of the IP interrupt. This insures that if the CPUs in the instance should attempt to execute at a priority below that of the IP interrupt, the IP interrupt will occur first and thus the CPU will see the "bugcheck" bit before any lower priority threads can execute. This insures the operating system instance will crash and not touch shared resources such as memory which may have been reallocated for other purposes when the instances were judged dead. As an additional or alternative mechanism, a console callback (should one exist) can be invoked to remove the instance. In addition, in accordance with a preferred embodiment, whenever an instance disappears or drops out of the APMP computer system without warning, the remaining instances perform some sanity checks to determine whether they can continue. These checks include verifying that all pages in the APMP database are still accessible, i.e. that there was not a memory failure.

Assignment of Resources After Joining

[0144] A CPU can have at most one owner partition at any given time in the power-up life of an APMP system. However, the reflection of that ownership and the entity responsible for controlling it can change as a result of configuration and state transitions undergone by the resource itself, the partition it resides within, and the instance running in that partition.

[0145] CPU ownership is indicated in a number of ways, in a number of structures dictated by the entity that is managing the resource at the time. In the most basic case, the CPU can be in an unassigned state, available to all partitions that reside in the same sharing set as the CPU. Eventually that CPU is assigned to a specific partition, which may or may not be running an operating system instance. In either case, the partition reflects its ownership to all other partitions through the configuration tree structure, and to all operating system instances that may run in that partition through the AVAILABLE bit in the HWRPB per-CPU flags field.

[0146] If the owning partition has no operating system instance running on it, its console is responsible for responding to, and initiating, transition events on the resources within it. The console decides if the resource is in a state that allows it to migrate to another partition or to revert back to the unassigned state.

[0147] If, however, there is an instance currently running in the partition, the console relinquishes responsibility for initiating resource transitions and is responsible for notifying the running primary of the instance when a configuration change has taken place. It is still the facilitator of the underlying hardware transition, but control of resource transitions is elevated one level up to the operating system instance. The transfer of responsibility takes place when the primary CPU executes its first instruction outside of console mode in a system boot.

[0148] Operating system instances can maintain ownership state information in any number of ways that promote the most efficient usage of the information internally. For example, a hierarchy of state bit vectors can be used which reflect the instance-specific information both internally and globally (to other members sharing an APMP database).

[0149] The internal representations are strictly for the use of the instance. They are built up at boot time from the underlying configuration tree and HWRPB information, but are maintained as strict software constructs for the life of the operating system instance. They represent the software view of the partition resources available to the instance, and may - through software rule sets - further restrict the configuration to a subset of that indicated by the physical constructs. Nevertheless, all resources in the partition are owned and managed by the instance - using the console mechanisms to direct state transitions - until that operating system invocation is no longer a viable entity. That state is indicated by halting the primary CPU once again back into console mode with no possibility of returning without a reboot.

[0150] Ownership of CPU resources never extends beyond the instance. The state information of each individual instance is duplicated in an APMP database for read-only decision-making purposes, but no other instance can force

in a one-to-one fashion.

[0158] An instance may specify a "context variable" which is to be associated with a region. If another instance attempts to attach to a region and does not specify the same context, an error is returned. This specification of a context variable may be used, for example, to associate a version number with the application. Additionally, an instance may specify a private context variable to be associated with the instance private data stored for a region. When the call-back routine is called, the instance can gather additional information about the region by obtaining the private context variable. The private context may be used, for example, to store a port number.

[0159] Shared memory can be borrowed by an operating system instance for use as instance private memory. Shared memory can be borrowed through the use of the shared memory API. Shared memory can be created, then used by only the local instance. This technique is useful if not all memory marked as shared is being used by the community member instances. The extra shared memory can be a pooled source of free memory. In other words, shared memory can be borrowed by the creation of a shared memory region. The pages in the shared memory region can be used by the local operating system for various purposes.

[0160] Private memory can be configured to be owned by the instance whose CPU(s) have fastest access to the memory. Nonuniform memory access is accommodated in the design's shared memory by organizing internal data structures for shared memory in groups according to the hardware characteristics of the memory. These internal data structures are called common property partitions. The shared memory API allows for memory characteristics to be specified by the caller. These characteristics can be expressed as nonuniform memory access properties such as "near" or "far".

[0161] The PFN database accommodates private memory and shared memory and reconfigured memory using a large array of page frame number (PFN) database entries. There is no physical memory behind a virtual array that describes pages that are private to another instance, nor corresponding to memory locations supported by memory boards that are missing from the system; nor corresponding to physical memory addressing holes. The layout of the PFN database suggests a particular granularity of physical memory. That is, in order to allocate and consume an integral number of physical pages for the PFN database that is to reside within each block of memory, physical memory should have a granularity as described below. The granularity of physical memory is chosen as the the smallest amount of memory that contains an integral number of pages and an integral number of PFN database entries. This is given by the least common multiple of the memory page size and the page frame number database entries, in quad words.

[0162] As described above, a creating instance, more specifically, the APMP computer system's initialization program, walks the configuration tree and builds management structures for its associated community's shared memory. In general, four hierarchical access modes provide memory access control. The access modes are, from the most to least privileged: kernel, executive, supervisor and user. Additionally, memory protection is specified at individual page level, where a page may be inaccessible, read only, or read/write for each of the four access modes. Accessible pages can be restricted to have only data or instruction access. Memory management software maintains tables of mapping information (page tables) that keep track of where each virtual page is located in physical memory. A process, through a memory management unit, utilizes this mapping information when it translates virtual addresses to physical addresses. The virtual address space is broken into units of relocation, sharing, and protection pages, which are referred to as pages. An operating system instance controls the virtual-to-physical mapping tables and saves the inactive parts of the virtual memory address space on external storage media.

[0163] Memory management employs, illustratively, a quad word page table entry to translate virtual addresses to physical addresses. Each page table entry (PTE) includes a page frame number (PFN) which points to a page boundary and may be concatenated with a byte-within-page indicator of a virtual address to yield a physical address.

[0164] Physical address translation is performed by accessing entries in a multi-level page structure. A page table base register (PTBR) contains the physical PFN of the highest level page table. Bits of the virtual address are used to index into the higher level page tables to obtain the physical PFNs of the base lower level page tables and, at the lowest level, to obtain the physical PFN of the page being referenced. This PFN Is concatenated with the virtual address byte-within-page indicator to obtain the physical address of the location being accessed.

[0165] As noted above, an instance may decide to join the operation of a community at any time, not necessarily at system boot time. When an instance decides to join the APMP system, it calls a routine DB_MAP_initial, which obtains the APMP data base pages from the configuration tree community node and maps the initial piece of the APMP database. If the configuration tree does not contain APMP database pages yet, the instance chooses shared memory pages to be used for the APMP database. The instance calls console code to write to the configuration tree in an asynchronous manner. After mapping the initial piece of the APMP database, it is determined as described above whether the instance is creating or joining the APMP system.

[0166] If the instance is the creator of the APMP system, the instance calls a routine, DB_allocate, to allocate the pages for the APMP database and to initialize the mapping information within a MMAP data structure. The MMAP data structure, which is discussed in greater detail below, is used to describe a mapping of shared memory. The routine DB_allocate does not unmap the initial piece of the APMP database. If the instance is a joiner of a APMP system, the

valid shared memory regions and an offset from the beginning of the shared memory data structure to the shared memory region tag array. The size of a shared memory region structure, and the offset from the beginning of the shared memory management data structure to the shared memory region array is also included.

[0175] Instance private memory data cells contain information about the shared memory management area in the APMP database. This information includes a pointer to the beginning of the shared memory data structure and the same descriptors as were described in relation to the shared memory data structure: the maximum number of shared memory common property partitions, maximum number of memory fragments in each shared memory common property partition, the size of one shared memory common property partition structure, a pointer to a shared memory common property partition array within the APMP database; a pointer to a shared memory list and a pointer to a shared memory region tag array within the APMP database. Additionally, the maximum number of shared memory regions, the size of one shared memory region structure, a pointer to a shared memory region array within the APMP database, and a pointer to the shared memory descriptor array in private memory are included.

[0176] When a shared memory common property partition (CPP) configuration area is initialized, the APMP database pages are excluded. Shared memory common property partitions support hot-swapping and non-uniform memory access by partitioning shared memory into partitions having common properties. Flags and routines are employed to indicate, for example, which non-uniform memory access unit a CPP is in, or which hot swappable unit a CPP is in, along with the range and location of memory pages within the unit. Each instance that is a member of an APMP system maintains data within its own private memory regarding each shared memory CPP that it is connected to. A lock structure is employed to synchronize access to the shared memory common property partition data structure. The lock is held when a partition is connecting to the shared memory CPP, when a partition is disconnecting from a shared memory CPP, when pages are being allocated from the shared memory CPP, or when pages are being deallocated to the shared memory CPP. Each shared memory CPP has a free page list, a bad page list, and an untested page list. Pages can be allocated from the free page and untested page lists and deallocated into the free page list and bad page list. The shared memory CPP page list links are maintained within the PFN database entries for the pages.

[0177] The shared memory lock is employed to synchronize the SHM_TAG array and an associated list of valid SHM_REG structures and to synchronize access to the list of free SHM_REG structures. The SHMEM lock must be held while reading or writing the SHM_TAG array, while manipulating a list of valid SHM_REG structures, or while manipulating the free SHM_REG list. Shared memory locks are ranked as follows: the highest order lock is an IPL 8 SMP spinlock, followed by the SHM_CPP lock, the SHM_REG lock and, finally, the SHMEM lock. For example, while holding the SHMEM lock, one can aquire a SHM_REG lock, a SHM_CPP lock and/or a SMP spinlock, in that order. Shared memory management functions can be called from kernel mode to get information about shared memory. The SHMEM lock has a ranking relative to other locks such that no deadlocks occur.

B. MIGRATING RESOURCES IN A MULTI-PROCESSOR COMPUTER SYSTEM (FIGS. 9-10E)

[0178] In accordance with an aspect of the invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein a single physical machine is subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each individual instance has the resources it needs to execute independently, but instances cooperate to migrate resources from one partition to another. In accordance with the principles of this aspect of the invention, the migration can be initiated and carried out under control of the operating system instances "on the fly" without intervention of the system administrator. Alternatively, a system administrator can reconfigure the system.

In accordance with one embodiment, resource migration is carried out under a "push" model in which resources are controlled by an owning partition and must be released by that partition before they can be migrated to another partition. In accordance with this model, a first operating system instance which requires a resource first requests the resource from a second instance. In response to this request, the second instance determines whether it can spare the resource, and if so, begins to bring the resource into an idle state. The resource is transferred when the second instance stops using the resource. Also, in accordance with the "push" model, the request for the resource migration may be initiated from an operator running a program on the second instance or from policy management software which initiates the request.

Runtime Migration of Resources

[0179] After an APMP system is running, resources initially allocated to one partition can move, or migrate, to another partition. This migration may take place under control of a system administrator or may be initiated by an operating instance without system administrator participation. Migration is accomplished by causing the owning operating system instance to stop using a resource or by a console program taking action to stop using a resource. The configuration tree is then altered to transfer ownership of the resource to another operating system instance. The new operating

a resource migration, the HWRPBs and configuration trees are modified in concert and atomically to prevent any instance from obtaining an incorrect view of the configuration. In a similar fashion, a "hot swap" must also be accompanied by atomic and coordinated modifications to the HWRPBs and configuration trees.

[0190] When a resource is de-assigned, the owning operating system may choose to de-assign the owner field, or both the owner and current_owner fields. The decision of which field to de-assign is based on the ability of the owning operating system to discontinue the use of the resource prior to de-assigning ownership. In the case where a reboot is required to relinquish ownership, the owner field is cleared, but the current_owner field is not changed. When the owning operating system instance reboots, the console program in the partition can be designed to clear any current_owner fields for resources that have null owner fields during initialization.

[0191] In the case where the resource is shared among operating system instances which are part of a community, the operating system instances must cooperate in order to de-assign the resource. This de-assignment is managed by the instances which are part of the community.

[0192] Ownership of a resource is altered by changing the configuration tree. Certain rules must be followed when the configuration tree is altered. They include the following:

1) If the config field of a configuration tree node points to the tree hardware root node, the corresponding resource is capable of being assigned independently to any community or partition.
2) If the config field of that node does not point to the tree hardware root node, when the resource is assigned to a partition, all descendants of the corresponding tree node must be modified so that they are assigned to the same partition and all tree nodes in the parent chain from the tree node being altered to the config node must also have the same Partition, or Community owner. The config node must be an ancestor of the node being changed and reachable by following the parent pointers.

[0193] When the ownership of a hardware node is given to a Community, all descendants must either be owned by the community, a partition in the community, or unowned. If the config pointer is not the hardware root, then all nodes in the parent chain from the node being altered to the config node must also have the same Community owner. The config node must be an ancestor of the node being changed and reachable by following the parent pointers.

[0194] Hardware components, for example a CPU, may be free to operate independently. In this case, the config pointer specifies the hardware root. However, components may also be made up of other components that cannot operate independently. For example, the I/O Controllers on a PCI bus might not be separable from the PCI bus.

[0195] Some hardware components, such as memory may be shared by multiple partitions. This is expressed by the owner field pointing to a community. Partitions in the community, may then share access to the component. The descendants of the hardware component that is shared may specify an owner that is a descendent. For example, a memory subsystem may be owned by the community. In this case, the memory controller, which expresses the hardware aspects of the memory, including error handling, is owned by a partition in the community, and the memory descriptor node may be owned by the community, and its fragments owned by both communities (shared) and by partitions (private).

Alternative Embodiments.

[0196] Other migration scenarios are possible. For example, a system manager could direct the first operating system instance to quiesce a resource and then place the resource into the "unassigned" state. Some time later, a second instance could discover a need for the resource and remove it from the pool of unassigned resources. Alternatively, the system manager could direct the second instance to allocate the resource. Alternatively, system policy might cause a batch job to run at a predetermined time on the first instance that would unilaterally quiesce and transfer some resource to a second instance. In both of these latter examples, there is no explicit communication between the instances regarding the resource, although there is clearly some human intervention or policy that coordinates the overall usage of resources.

[0197] In yet another approach, instances would allocate a resource from the unassigned pool when their needs require it. When they are finished with the resource, they return it to the unassigned pool. If the pool is empty, an instance has to operate with its then-existing allocation of the resource. So, we have a common pool, but no request/release protocol is required between instances. Other aspects, such as the specific instructions utilized to achieve a particular function, as well as other modifications to the inventive concept are intended to be covered by the appended claims.

to the point where the operating system instances present a single cohesive entity to the network. For example, the operating system instances may appear to the user as an OpenVMS Cluster.

[0207] Even though some operating system instances are sharing resources, one or more operating system instances can execute in total software isolation from all others. An instance that exists without sharing any resources is called an independent instance and does not participate at all in shared memory use. More particularly, neither the base operating system, nor its applications, access shared memory. The inventive computer system could consist solely of independent instances, such a system would resemble traditional mainframe style partitioning.

[0208] Figure 12 shows how the inventive system can be configured to support a "shared nothing" model of computing. In this example, three partitions, 1206, 1210 and 1214, have been created within a single machine 1200, each running an instance of an operating system. The available twelve CPUs have been split arbitrarily equally between the partitions in this example, 1206, 1210 and 1214, as illustrated. The available memory has been divided into private memory and assigned to the instances. In Figure 12, private memory divisions 1216, 1218 and 1220 are illustrated. Both code and data for each instance is stored in the the private memory assigned to that instance. Although the memory, 1216, 1218 and 1220 has been illustrated in Figure 12 as divided equally, the inventive architecture supports arbitrary division of memory between instances. Thus, if a partition has large memory requirements, and another that has limited memory needs, the system can accommodate both, in order to maximize use of available memory.

[0209] Each partition also has private I/O controllers/disks illustrated as I/O portions 1204, 1208 and 1212. Just as memory is arbitrarily divided, the same characteristic is true of I/O circuitry. In the "shared nothing" arrangement illustrated in Figure 12, the available I/O resources are private to each partition, but do not have to be evenly divided. The partitions 1206, 1210 and 1214 can be networked together over a physical link 1202 within the machine, and that link can be extended off the machine to other computers.

[0210] What has just been created is the equivalent of three physically separate machines networked together. The difference between a conventional computer system and the inventive system is that, instead of having three physical boxes, there is only one box. Also, the exact configuration of each partition need not be determined until machine deployment. Another unique feature of the inventive system is the ability to dynamically configure the number/size of the partitions after machine deployment.

[0211] Figure 13 illustrates the inventive system configured as a shared-partial model in which the partitions share memory. In Figure 13, elements which correspond to elements in Figure 12 have been given corresponding numerals. For example, machine 1200 in Figure 12 has been designated as machine 1300 in Figure 13. As before, each partition 1306, 1310 and 1314, has its own private memory section, 1316, 1318 and 1320, respectively, in which code and data for that instance are stored. However, in this configuration, there is also a shared memory section 1322 in which data and/or code accessible by all instances 1306, 1310 and 1314 is stored. The three instances, 1306, 1310 and 1314, are also networked together by interconnect 1302.

[0212] The advantage of the configuration illustrated in Figure 13 is that large shared cache memories (for example, databases or file systems) can now be created and used jointly by several instances. The system also has the advantage that an instance of an operating system can deactivate or leave the configuration, and upon rejoining, can re-map into a still active cache memory. As cache memories become larger, this ability to remap into an existing memory is extremely important since it is very time-consuming to load all entries in very large cache memories into a private memory space.

[0213] Figure 14 illustrates the inventive system configured to operate as a "shared everything" computer system. As with Figure 13, elements in Figure 14 that correspond to similar elements in Figure 12 and 13 have been given corresponding numerals. Each partition, 1406, 1410 and 1414, still has its own private memory, 1416, 1418 and 1420, respectively, in which code and data for that instance are stored. There is also a shared memory section 1422 in which data is stored. The three instances are networked together by interconnect 1402, but there is also a storage interconnect 1424 and a cluster interconnect 1426. Configuring the system for shared everything computing also brings the following advantages:

1) a distributed lock manager (not shown) can use shared memory 1422 to store its lock cache (not shown), thus increasing lock performance.
2) a cluster interconnect 1428 can be placed in shared memory 1422, instead of using independent hardware, thus increasing cluster communication performance if the instances 1406, 1410 and 1414 are clustered.
3) partitions can be created for highly specialized functions. For example, a partition could be created without I/O controllers (not shown), effectively making it a "compute engine".

[0214] With the inventive system, it is possible to run all three computing models illustrated in Figures 12, 13 and 14 within a single computer box by appropriate configuration of partitions. That is, some partitions can operate as "shared nothing" computers. A separate group of partitions can operate as a "shared partial" computing system and a still further group of partitions can operate as a "shared everything" computing system. Further, it is possible for a given

control of the instance, the first bit is set to a second assertion level. With a bit representative of each of the CPUs, this bitvector then provides designations for each of the CPUs indicative of which are under control of the instance. Similarly, other bitvectors also provide designations for each of the CPUs, those designations indicating, for example, which CPUs are compatible for operation with the instance, which are available to the instance for SMP operation, and which would be allowed to join SMP processing activities immediately after being initialized. In this way, each of the instances may individually track all of the processing resources and what their operational statuses are relative to the instance.

[0222] In an alternative embodiment, designations indicating the operational statuses of processing resources relative to the instances of the system are maintained in a storage area accessible to all the instances. In particular, information regarding the compatibility of a processor with each of the different instances is provided. This allows each instance to identify whether a given processor might be appropriate for transfer to a particular instance.

Virtual Resource Management

[0223] In a preferred embodiment of the invention, the CPU resources for the computer system are arranged in a particular hierarchy relative to each instance. That is, the CPUs of the system are identified by each instance, and each instance categorizes the CPUs according to its own use or potential use of them. This is explained in more detail below.

[0224] In the preferred embodiment, each of the instances maintains a record of the CPUs in the system, categorizing them each in one of three sets: the "potential" set; the "configure" set; and the "active" set. From the perspective of a given instance, the potential set covers all of the CPUs which could possibly execute on that instance at any time. This typically includes all CPUs in the system except for those which are of a configuration or revision level which makes them incompatible with the instance and/or the partition on which it is running. Each instance makes a determination of those CPUs in the system that are compatible with it, and includes them in its potential set.

[0225] The configure set contains all of the CPUs that, for the given instance, are under that instance's control. That is, the configure set includes all the CPUs that are controlled (or managed) by the instance and which are currently participating, or are capable of future participation, in SMP operation. Once control of a CPU is acquired by the instance (indicated by the setting of the "current_owner" bit of that CPU in the per-CPU bits of the HWRPB for the partition on which the instance is running), there is a period during which it initializes itself to operation with the new instance. During this period, the CPU is not participating in SMP operation, but is nonetheless part of the configure set for that instance. Once the initialization is complete, the CPU makes a request to join SMP operation. Once it has joined, the new CPU is considered part of the active set as well. The active set includes all CPUs that are participating in SMP operation for the instance. A CPU in active mode is capable of pulling instructions from the instruction queue as part of the scheduling model of the instance.

[0226] Figure 16 is a schematic representation of several partitions 1600, 1602, 1604, and how the instance for each of the partitions has organized the CPUs into sets. Although this example uses only eight different CPUs, it will be understood by those skilled in the art that any number of CPUs may make up the system. In the Figure 16 example, each of the instances has identified each of CPUs 0-7 as being in its potential set. Thus, either these represent all of the CPUs in the system, or any other CPUs in the system are not compatible with any of the instances. Of course, other examples may exist where a CPU is in the potential set of one instance, but not in that of another, assuming the two instances have different compatibility requirements.

[0227] The configure sets for the instances on partitions 1600, 1602, 1604 are different for each instance. This must always be the case, since only one instance can have control of a CPU resource at any given time. As shown, CPU 0, CPU 2, CPU 3 and CPU 7 are in the configure set of the instance running on partition 1600. CPU 1 and CPU 5 are in the configure set of the instance running on partition 1602. Finally, CPU 4 and CPU 6 are in the configure set of the instance running on partition 1604. Thus, these CPUs are controlled, respectively, by these different instances.

[0228] The CPUs in the active sets of the three instances running, respectively, on partitions 1600, 1602, 1604 are the same as those in the configure sets for those instances, With the exception of CPU 2, which is in only the configure set of the instance of partition 1600. In this case, it may be assumed that CPU 2 has recently been moved to the control of the instance on partition 1600, and is going through an initialization stage before joining SMP operation. Once it does join the other CPUs of the system in actual processing activities, it will become part of the active set for the instance of partition 1600.

[0229] In the preferred embodiment, each of the instances keeps track of the status of its ownership rights in the CPUs via groups of bits, or "bitvectors." The bitvectors of each instance are used for tracking CPU participation in that instance's potential set, configure set and active set, respectively. An example of this is shown schematically in Figure 17, which depicts schematically the bitvectors for each of the potential, configure and active sets of the instance running on partition 1600.

[0230] As shown in Figure 17, for each of the groups of bits in question, the assertion level of a bit indicates the

E. DYNAMICALLY SHARING MEMORY IN A MULTIPROCESSOR SYSTEM (FIGS. 1-8B)

[0236] In accordance with a further aspect of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.

[0237] More particularly, software logically, and adaptively, partitions CPUs, memory, and I/O ports by assigning them together. An instance of an operating system may then be loaded on a partition. At different times, different operating system instances may be loaded on a given partition. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. The resulting adaptively-partitioned, multi-processing (APMP) system exhibits both scalability and high performance.

[0238] The execution environment for a single copy of an operating system, or instance, is referred to as a partition. A community is a grouping of partitions which can share resources. Memory may be private to a particular partition or may be shared by partitions within a community. When an APMP computer system is formed the creating instance reads a configuration tree and builds management structures for the shared resources, including memory, owned by the community. A single system may have one or more communities, each with its own representation within the configuration tree.

[0239] Configuration software selects a group of shared memory pages to be the synchronization point for an APMP computer system. These pages contain information used to determine whether other instances are active members of the APMP computer system. An indication of the location of these synchronization pages is stored within the configuration tree. As an instance joins the APMP computer system, it uses the information within the configuration tree to map to the shared pages. From the contents of the shared pages, the instance can determine whether it is joining an active APMP computer system or it is creating a APMP computer system. If the instance is creating a APMP computer system, it can reconfigure the pages used to synchronize the APMP computer system by modifying the configuration tree. In this way, physical memory that was previously used as a synchronization point may be removed from the system.

[0240] Shared memory may be organized in groups, referred to as common property partitions, according to the hardware characteristics of the memory. Shared memory may be assigned to regions which can be mapped simultaneously by one or more operating system instances. Shared memory may also be mapped by applications running on one or more operating system instances. Shared memory may be "borrowed" by an operating system instance for use as the instance's private memory. Additionally, non uniform memory access is accommodated, in the case of private memory, by allowing the private memory to be owned by the instance whose CPUs have the fastest access to the memory.

F. RECONFIGURING MEMORY IN A MULTIPROCESSOR SYSTEM WITH SHARED MEMORY (FIGS. 19-21)

[0241] In accordance with a further aspect of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.

[0242] More particularly, software logically, and adaptively, partitions CPUs, memory, and I/O ports by assigning them together. An instance of an operating system may then be loaded on a partition. At different times, different operating system instances may be loaded on a given partition. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. The resulting adaptively-partitioned, multi-processing (APMP) system exhibits scalability, flexibility, and high performance.

[0243] Memory may be reconfigured into or out of a partition or community under software control and hardware hot in-swapping or out-swapping are supported. In general, memory may be in one of three states: private, shared, or unowned. Memory is private if it is "owned" by a single system partition. Memory is shared if it is owned by a community. A community is a collection of one or more partitions which may share resources. Otherwise, it is unowned. Memory

to avoid looping forever.

[0250] If there are no pages on the free, zeroed, untested, or bad page lists, the process proceeds from step 1906 to step 1910 where it is determined whether there are any pages on the modified page list. If there are no pages on the modified page list, the process proceeds to step 1914. On the other hand, if there are pages on the modified page list, the process proceeds to step 1912 where the pages are written to backing storage such as a system page file. From step 1912 the process proceeds to step 1926 and from there as previously described. On the other hand, if the process had proceeded to step 1914 from step 1910, in step 1914, it is determined whether any pages are process pages, i.e, pages where an application runs. Since each process may have multiple threads, each process has its own page table apart from other processes. If any pages are process pages, an AST which provides a mechanism for executing within the process' context to gain access to the process page tables, is sent to the process and the AST sets the process single threaded if necessary to synchronize access to the page tables. New page frame numbers are allocated for the pages and the contents of the pages are copied to the new page frame numbers. The old page frame numbers are placed on the instance's removed page list in step 916. From step 1916, the process proceeds to step 1926 and from there as described previously.

[0251] If it is determined in step 1914 that none of the pages are process pages, the process proceeds to step 1918 where it is determined whether any of the pages are part of a global section i.e., a set of private pages acesssed by several processes simultaneously. If any of the pages are part of global section, the process proceeds from step 1918 to step 1920. In step 1920, the name of the global section may be displayed to a user so the user can determine which application to shut down in order to free memory. Alternatively, an operating system which can track where pages are mapped, could suspend all processes mapped to the section, copy all pages, modify all process page table entries, and place the old page frame numbers on the removed page list. From step 1920, the process would then proceed to step 1926 and from there as described previously. If in step 1918 it is determined that there are no pages which are part of the global section, the process proceeds to step 1922 where it is determined whether any page is mapped into system address space. If none of the pages is mapped into the system address space, the process proceeds from step 1922 to step 1926 and from there as previously described.

[0252] On the other hand, if there are pages mapped into the system address space, the process proceeds to step 1924, where those pages that are read only are copied. Once copied, the page is placed on the removed page list. Read/write pages are copied only while all CPUs within the instance are temporarily blocked from execution such that they do not change the contents of the page. One page is copied at a time and copied pages are placed on the remove page list. After migration in step 1924, the process proceeds to 1926 and from there as previously described.

[0253] In order to remove shared memory, a new list of page frame numbers is established within each shared memory common property partition data structure, i.e., within each SHM CPP structure within the APMP data base. The list is called the common property partition removed page list. These lists are temporary holding places for all pages that are ready to be removed. Only when all pages within the 8 MB range are located on the appropriate remove page list, can the pages be removed. At any point in time, if it is determined that the pages cannot be removed while the sharing set (the set of instances which share resources, such as memory) is operational, the sharing set can be dissolved, the pages marked unowned, and the sharing set reformed. The removal operation can be abandoned at any time by moving the pages from the common property partition removed page list to the free page or bad page lists depending on the previous state of the page, i.e., to the free page list if it had been on the free page list and to the bad page list if it had been on the bad page list.

[0254] The process of removing shared memory is illustrated in the flowchart of Figure 20 with the process beginning in step 2000 and proceeding from there to step 2002 whereby the first unremoved page in the range of pages to be removed is located. From step 2002 the process proceeds to step 2004 where it is determined whether any pages within the page range to be removed have outstanding I/O. Those pages that have outstanding I/O are skipped and the process returns to step 2002 to locate the first unremoved page, as previously described. To allow the system to perform other work, a system may want to stall at this point waiting for I/O to complete.

[0255] If no pages have outstanding I/O, the process proceeds from step 2004 to step 2006 where it is determined whether any pages are part of the APMP database. If any of the pages are a part of the APMP database, an error is returned to the user in step 2008. In such a case, the sharing set must be reformed with the APMP database on different pages. This can be accomplished by dissolving the sharing set, marking the old APMP database pages as unowned, choosing a new set of APMP database pages, then recreating the sharing set. The process proceeds from step 2008 to finish in step 2030.

[0256] The page frame number database entry pages describing those pages are unmapped and freed to the common property partition free page list, the PMAP arrays that describe shared memory are updated to reflect the change and the console is called to mark the page unowned. From step 2028, the process then proceeds to finish in step 2030. If, in step 2026, the process had determined that all pages were not on the shared memory common property partition removed page list, the process returns from step 2026 to step 2002 and proceeds from there as described previously. If in step 2006, the process determines that none of the pages are part of the APMP database, the process proceeds

1) Set the ID field in the instance's partition node to the current ID.

[0265] During a sharing set exit:

1) Call a routine which detaches from all shared memory regions.
2) Clear the ID field in the instance's partition node in the config tree.

[0266] When an I/O is initiated, the routine shm_reg_incref is called for each page on which I/O will be performed. When the I/O is completed, the routine shm_reg_decref is called for each page.

Routine shm_reg_incref:
Input: Address of PFN database entry for page
Read the shared memory region id from the PFN database entry.
Obtain the SHM_DESC address in private memory for this region.
Increment I/O refcnt.
Routine shm_reg_decref:
Input: Address of PFN database entry for page
Read the shared memory region id from the PFN database entry.
Obtain the SHM_DESC address in private memory for this region.
Decrement I/O refcnt

Routine shm_reg_create:
Inputs: tag

virtual length
physical length
virtual mapping information
plus additional information

Output: shared memory region id

[0267] Search for a SHM_REG structure in shared memory whose tag matches the tag supplied.

[0268] If no such SHM_REG structure is found:

- Allocate a new SHM_REG structure
- Acquire the SHM_REG lock
- Set the "init in progress" bit in the SHM_REG structure
- Allocate shared memory pages for the region
- Clear the "init in progress" bit in the SHM_REG structure
- Release the SHM_REG lock

Set the bit in the SHM_REG attached bitmask for this instance.
Map to the shared memory region using the virtual mapping information supplied.
Routine shm_reg_delete:

Input: shared memory region id

[0269] Obtain the SHM_DESC address in private memory for this region.

If the I/O refcnt field is non-zero, return an error.
Unmap the shared memory region.
Obtain the SHM_REG address in shared memory for this region.
Clear the bit in the SHM_REG attached bitmask for this instance.
If the attached bitmask has more bits set, return.
If the SHM_REG attached bitmask has no bits set:

- Acquire the SHM_REG lock

If the attached bitmask has no more bits set

- Call shm_reg_delete

Loop to the next SHM_REG structure
After all SHM_REG structures have been processed, return.

[0273] To choose the initial APMP database pages, the routine shmem_config_APMPDB is called by APMPDB_map_initial to choose the intial set of APMPDB pages.

[0274] Data structures:

[0275] The community node in the config tree contains a 64-bit field, called APMPDB_INFO, which is used to store APMPDB page information. The first 32-bits, APMPDB_INFO[31:0], is the low PFNof the APMPDB pages. The second 32-bits, APMPDB_INFO[63:32], is the page count of APMPDB pages.

[0276] Each instance keeps an array in private memory called the "shared memory array." Each element in the array contains a shared memory PFN and a page count. The entire array describes all shared memory owned by the community that this instance is a part of.

[0277] The configuration tree may contain tested memory bitmaps for shared memory. If the configuration tree does not contain a bitmap for a range of memory, the memory has been tested and it is good. If a bitmap exists for a range of memory, each bit in the bitmap indicates whether a page of shared memory is good or bad.

[0278] A value, MAX_APMPDB_PAGES, is set to the maximum number of pages required to initialize a system. This number should be smaller than the granularity of shared memory. MAX_APMPDB_PAGES should be a small number to increase the chances that contiguous good memory can be found for the initialization of the APMP database.

Console callback routine set_APMPDB_info:

Inputs: new_APMPDB_info Bits[31:0] = first APMPDB page frame number Bits[63:32] = number of pages specified for the APMP database
old_APMPDB_info - value read from the GDMB_INFO field in the community node.

Outputs: None.
Status:

Error = Value in APMPDB_INFO does not match old_APMPDB_info

Success = APMPDB_INFO has been updated with new_APMPDB_info

[0279] This routine may be more complex if multiple copies of the configuration tree are maintained by a console.

1. Read APMPDB_INFO from the community node
2. If APMPDB_INFO does not equal old_APMPDB_info, return an error
3. Store new_APMPDB_INFO into APMPDB_INFO with an atomic instruction

[0280] A routine, SHMEM_config_APMPDB, is used to configure the APMP database. The routine provides the first APMP database page frame number and the number of pages specified for the APMP database. The routine proceeds as follows:

(1) Obtain a pointer to the community node within the configuration tree.
(2)Traverse the configuration tree creating the shared memory array. If there is no shared memory, return an error.
(3) Read the APMPDB_INFO field
(4) If APMPDB_INFO field is non-zero

Set PAGES to APMPDB_INFO
Search the shared memory array to ensure that pages PFN through PFN+PAGES-1 are in shared memory.

If these pages are in the shared memory array:
if a tested memory bitmap exists, check the bitmap to ensure

that these pages are not marked bad

G. FAILURE RECOVERY IN A MULTI-PROCESSOR COMPUTER SYSTEM (FIGS. 22-32)

[0282] In accordance with a further aspect of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.

[0283] More particularly, software logically, and adaptively, partitions CPUs, memory, and I/O ports by assigning them together. An instance of an operating system may then be loaded on a partition. At different times, different operating system instances may be loaded on a given partition. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. The resulting adaptively-partitioned, multi-processing (APMP) system exhibits both scalability and high performance.

[0284] In the present invention, the individual instances each maintain a separate record of all of the processing resources of the system. Each of the instances categorizes the processors based on their respective operational status relative to the instance. In a preferred embodiment, an instance maintains records of whether each CPU is compatible for operation with the instance, whether it is under the control of the instance and whether it is available for SMP operation within the instance. These different operational statuses represent a hierarchical categorization of the CPUs of the system, and the system is adaptible to additional categories.

[0285] In the preferred embodiment, the membership of the CPUs in any of the different categories of operational status is recorded by maintaining a bitvector for each category, at least one bit of each bitvector corresponding to the membership status of one of the CPUs in that group. Typically, each set has one bit for each of the CPUs such that, for example, a bitvector indicative of CPU control by the instance has a first bit set at a first assertion level if a first corresponding CPU is under control of the instance. If the CPU is not under the control of the instance, the first bit is set to a second assertion level. With a bit representative of each of the CPUs, this bitvector then provides designations for each of the CPUs indicative of which are under control of the instance. Similarly, other bitvectors also provide designations for each of the CPUs, those designations indicating, for example, which CPUs are compatible for operation with the instance or which are available to the instance for SMP operation. In this way, each of the instances may individually track all of the processing resources and what their operational status is relative to the instance.

[0286] The invention may also include a means for automatically moving a processor from a first instance to a second instance in an instance failure situation. Such a movement can use the execution of an instruction by the moving processor. When a move is initiated, the migrating processor stores its current hardware state, and is capable of loading a hardware state that it held during a previous execution within the second instance. Thus, the processor resumes operation in the second instance from where it left off previously. If there is no stored hardware state in the instance to where the processor is migrating, it is placed in an initialized state.

[0287] In a preferred embodiment of the invention, the system responds to a failure within a first instance by not only shutting down the operating system running on that instance, but by first migrating out processing resources from the first instance to destination instances. That is, the control for a given resource associated with the first instance is transferred to a second instance without operator intervention. The system may have an instance that has no processing duties prior to failure of the first instance, but which serves as a backup to the first instance. Each CPU migrates to the destination instance, and preferably will take over the same functions that it undertook with the first instance. Each instance preferably stores the destination instance IDs in an array organized by CPU ID. Thus, upon the occurrence of a failure within the first instance, each processor under the control of the first instance can look to its associated array element to determine the ID of the instance to which it is to migrate.

[0288] The migration of processors from the first instance following a failure involves just secondary processors. The primary processor for that instance ensures that all of the secondaries migrate themselves, or that a separate migration routine is invoked to migrate any processors that may not be fully participating in SMP processing activities. The primary processor then stores all of the data in memory controlled by the first instance to a new memory location, after which it shuts itself down. Once the primary CPU of the first instance is shut down, the console program identifies that fact and sets a flag in shared memory indicating that the primary CPU of the first instance is in "console mode." The console then sends an interrupt to each of the other instances in the system, causing them to become aware of the setting of the flag. Meanwhile, processing is resumed on the backup instance. In certain cases, CPUs migrating from an instance that has experienced a failure may migrate to different destination instances. Allowing different destination instances for the different resources allows for flexibility in redistributing the resources of the system following a failure in one of the instances.

system, each ID identifying a destination instance (or partition) to which the CPU with which it is associated should migrate when its current host instance enters a failure mode. The arrangement of this information is depicted schematically in Figure 25, in which different sections of the ID portion of the memory are shown as each containing a destination ID. The adjacent CPU identifications associate each of the destination IDs with the appropriate CPU. Thus, when a failure is detected, the appropriate memory location may be examined to determine the target instance for a given CPU, and this information may be used to undergo a controlled migration to the new instance. For example, with the arrangement of partitions shown in Figure 24, many if not all of the secondary CPUs of partition 2400 could have the backup partition (partition 2402) designated as their target. Thus, if the instance on partition 2400 was to undergo a failure, these CPUs would migrate from partition 2400 to partition 2402. However, those skilled in the art will recognize that, in certain circumstances, it might be desirable to send one or more of the CPUs of partition 2400 to a different partition altogether. Furthermore, with the virtual resource functions of the instance, the newly acquired CPU can be brought into the active set automatically, for example, by making use of the autostart set described above.

[0297] During a failure of an instance, the system follows a series of steps designed to minimize the loss of usage of the resources of the failed instance and, in some cases, to attempt to transfer all of the operations being undertaken with the instance on the failed partition to a new instance and partition as quickly as possible. These steps are generally described in the flowchart of Figures 26A and 26B.

[0298] The sequence of events following a detected failure begins at starting step 2600 shown in Figure 26A. After the failure is detected, the operating system instance where the failure occurred takes control of the execution context of all the CPUs in its active set (step 2602). The remaining system shutdown procedures are pursued by the primary CPU of the partition, while the secondary CPUs undertake any procedures necessary for their migration. In step 2604, the primary CPU instructs the active secondary CPUs of the failed instance to each dump its processing context into a selected region of the system shared memory, after which they are instructed to begin the migration process. Once all of the active secondary CPUs have dumped their processing contexts, the primary CPU proceeds to the next action (step 2606). Since some of the secondary CPUs might not be in active mode, the primary CPU must confirm that each of the secondaries is proceeding with the failure recovery process. If there are any of the secondary CPUs which are in the configure set but not in the active set of the instance in question, the primary CPU will invoke a separate migration function in step 2608 for moving the stopped CPUs to any identified destination partition.

[0299] The failure recovery process is continued in Figure 26B, the continuity between Figure 26A and 26B being indicated by connector node "A". In step 2610, the primary performs a dump of the data stored in the memory controlled by the failing instance, copying it to a new location. As shown in step 2612 of Figure 26B, the primary CPU of the failed instance then sends a console "callback" instruction which indicates that it has entered console mode, and causes the console program to perform several different tasks. One of these tasks is to increment the system incarnation count, which is changed every time there is a change in the configuration tree. The console also initiates the generation of an interprocessor interrupt (IPINT), which causes all of the other instances to look at the incarnation count, notice it has changed, and examine the configuration tree, identifying the change in the state of the failing instance in the process. Furthermore, the console will set an appropriate state code in the per-CPU data bits for the primary CPU once it sees the primary shut down (i.e. enter console mode) without the possibility of returning to functioning on the present instance. The primary CPU entering console mode, and causing the console to set the aforementioned state code is shown in step 2614. At that point, the partition is completely disabled, and the failure steps are complete.

[0300] Once a secondary CPU (in the active set) is instructed by the primary CPU to begin a failure migrate, it undergoes a sequence of steps generally outlined in Figure 27. First, the secondary CPU dumps its processing context (step 2700), as instructed by the primary CPU. It then polls the appropriate portion of the memory array containing the failover IDs to determine where it is to migrate, if at all (step 2702). Once the target ID is retrieved, the processor compares it to the ID of the present partition (step 2704). In certain cases, it may be desirable not to migrate a CPU in the case of a failure on that partition. In such a case, the target ID loaded for the CPU in question will be the ID of the present partition. Thus, if the CPU determines in step 2704 that the target ID and the ID of the present partition are the same, it simply enters console mode, and the process terminates. However, if the target ID is different that the present ID, the CPU will undergo a migration to the new partition, using one of the provided methods of migration, after which the process is complete.

[0301] In one variation of the above embodiment, the interrupt handler may also be charged with notifying specific applications of the failure. In this alternative embodiment, the IPINT handler provides a registration process by which specific applications may "register" for notification of a particular event. Using this mechanism, any application could be notified of the change in the configuration tree, and could determine that a failure had occurred. If the application was a copy of an application that was running on the failed instance, this would provide a trigger to the application on the backup instance to begin processing activities in place of the application that was running on the failed instance.

an particular partition (as shown tested in step 2804), the CPU is then initialized on that partition (step 2806). Afterwards, the CPU itself sets its available bit (step 2808), and requests to join SMP operation on the partition (step 2810). The assignment process is then complete.

[0311] If, in step 2804, it is determined that there is no valid code indicating ownership of the CPU by any given partition, the CPU joins an HWRPB context as directed by the console (step 2805). That is, it is directed by the console to join an arbitrary partition. However, the CPU does not set its available bit and, as a result, it will not be expected to join SMP operation. It then enters a waiting state, during which it continuously polls its owner field (steps 2807 and 2809). When the owner field is filled with the code for a valid partition, the CPU locates that partition and, if necessary, migrates to it (step 2811). It then sets its available bit (step 2808) and requests to join SMP operation on the new partition (step 2810). This partition remains the "permanent owner" of this CPU until a change is made in its owner field.

[0312] The permanent ownership of a CPU may be changed with interaction by the console. This process is demonstrated in the flowchart of Figure 29, beginning in step 2900. In step 2902, the operating system instance of the partition on which the CPU in question is running executes a HALT instruction with a HALT_REQUESTED code of DEASSIGN. That is, a halt code is loaded into the HALT_REQUESTED bits of the per-CPU STATE_FLAGS field that indicates that the CPU is to be deassigned. The CPU responds to this code in several ways. It clears its owner field (step 2904) and its current_owner field (step 2906) in the CPU node. It also clears its available bit (step 2908). The CPU then begins examining its owner field waiting for a valid partition code to be placed in it. That is, the CPU proceeds to step 2802 of Figure 28, and undergoes the assignment process described above. This is demonstrated by connector "A" shown on both Figures 28 and 29.

[0313] When a secondary CPU is in console I/O mode, its deassignment may also be initiated by the primary CPU of the partition sending a DEASSIGN message to the secondary CPU's RX buffer. This causes the CPU to initiate the set of steps shown in Figure 29.

[0314] The permanent ownership characteristic of a CPU provides important information that is particularly useful for initial powering of the system. As the configuration tree is constructed, the permanent ownership of each CPU is used to construct the partitions. Later, the current_owner field (which designates the partition that controls a given CPU) is used to keep track of where each CPU is executing at any given time. However, the permanent ownership information is retained to allow for future use in reorganizing the system, or for future reinitializations of the system. Accordingly, the permanent ownership data is stored in nvRAM (i.e., non-volatile memory) while the temporary ownership data is stored in volatile memory.

I. MIGRATING PROCESSORS IN A MULTI-PROCESSOR COMPUTER SYSTEM (FIGS. 22A-27)

[0315] In accordance with a further aspect of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.

[0316] More particularly, software logically, and adaptively, partitions CPUs, memory, and I/O ports by assigning them together. An instance of an operating system may then be loaded on a partition. At different times, different operating system instances may be loaded on a given partition. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. The resulting adaptively-partitioned, multi-processing (APMP) system exhibits both scalability and high performance.

[0317] The invention can include a means for moving a processor from a first partition to a second partition. Such a movement requires the execution of an instruction by the moving processor, so that its acquiescence to the move (and that of the operating system instance on which it is running) is ensured. When a move is initiated, the migrating processor stores its current hardware state, and loads a hardware state that it held during a previous execution within the second partition. Thus, the processor resumes operation in the second partition from where it left off previously. If there is no stored hardware state in the partition to where the processor is migrating, it is placed in an initialized state.

[0318] The present system has interaction between the partitions that allows a processor to migrate from one partition to the other without requiring a reboot of the entire system. Software running on its current partition, or a primary processor in its partition, can provide the processor to be moved with a request that it initiate a migration operation. Such a migration may occur with or without interruption of the operating system in which it resides. That is, the processor may simply be quiesced and reassigned while the rest of the system continues to operate, or the resources in its partition may be halted a console program is invoked to coordinate the move.

[0327] When a CPU is executing under the direction of the operating system, the migration process is referred to as "halt" migration, because the CPU must be halted prior to being reassigned via the PAL_MIGRATE instruction. The steps necessary for halt migration are shown in Figures 30A and 30B.

[0328] Beginning in step 3000, the migration routine proceeds to step 3002 where the migrating CPU, in response to an instruction from the primary CPU to migrate, places a special HALT_MIGRATE code in the REASON_FOR_HALT per-CPU field of the HWRPB. Next, in step 3004, the CPU places the destination partition ID into the REASON_FOR_HALT field. The CPU then executes a HALT instruction (step 3006). The operating system is responsible for quiescing operation of the CPU, and ceasing SMP participation of the CPU prior to executing the HALT. The actual performance of these substeps is operating system specific and conventional.

[0329] When a CPU executes a HALT with a HALT_MIGRATE code in the REASON_FOR_HALT per-CPU field, control is transferred to the console HALT entry point as indicated in step 3008. The console then obtains the partition ID of the destination from the REASON_FOR_HALT per-CPU field (step 3010) and clears the REASON_FOR_HALT field as illustrated in step 3012. The console next determines whether the HALT_MIGRATE operation is valid in step 3014. If it is valid, the console clears the current_owner field in the CPU node of the configuration tree in step 3016, and executes a PAL_MIGRATE instruction in step 3018, after which the routine ends in step 3020. If, in step 3014, the console determines that the HALT_MIGRATE operation is not valid (for example, an invalid partition ID is specified), the console performs the steps in Figure 30C.

[0330] As shown in Figure 30C, a determination by the console that a HALT_MIGRATE operation is invalid results in its first clearing the REASON_FOR_HALT per-CPU field (step 3022) and remaining in a HALT state in the current partition. In step 3024, the console places a STARTREQ message into the CPU's TX buffer in the per-CPU slot and, in step 3026, sets its TXRDY bit in the HWRPB. Next, the console signals the primary CPU in the partition by means of an interrupt as set forth in step 3028. The console then polls the RXRDY bit in the HWRPB waiting for a command, such as START, DEASSIGN or MIGRATE to begin operation as set forth in step 3030. The routine then ends. When the HALT_MIGRATE operation is not validated, the console does not clear the available bit in the per-CPU flags and does not clear the current_owner field in the CPU node in the configuration tree.

[0331] When a CPU is already in console mode, the steps for migrating it are slightly different than those described above. A CPU which is in console IO mode, with the available bit set, and current_owner field in the configuration tree set will poll the per-CPU RX buffer waiting for a command from the primary CPU running the operating system. The steps followed to migrate this CPU are shown in Figure 31.

[0332] The method shown begins in step 3100 and proceeds to step 3102, in which the primary places a MIGRATE command in the RX buffer of the CPU to be migrated. The primary then sets the CPU's RXRDY bit to alert it to the presence of the migration instruction (step 3104). The current_owner field of the migrating CPU is then cleared by the console, in preparation for it reassignment (step 3106). The PAL_MIGRATE instruction (Figure 22A-22B) is then called (step 3108), after which the process ends in step 3020.

[0333] When a CPU is executing under its operating system instance, a method of migrating it which doesn't require immediate designation of the destination partition (as in the halt migration discussed above) is to first deassign the CPU, and to subsequently reasssign it to a different partition. This method is demonstrated in Figures 32A and 32B.

[0334] The method starts in Figure 32A at step 3200, and proceeds to step 3202, in which the primary CPU in the partition in which the migrating CPU resides places a DEASSIGN code in the RX buffer of the migrating CPU. The CPU is then instructed by the primary to execute a HALT instruction, which it does (step 3204). The presence of the DEASSIGN code when the HALT is executed causes the CPU to clear the owner field in the configuration tree, as well as the AVAILABLE bit and the current_owner field. The CPU then executes in console IO mode, joining the HWRPB context as directed by the console.

[0335] Being unassigned, the CPU continuously polls its owner field (steps 3208 and 3210). When the owner field is filled with the code for a valid partition, the CPU locates that partition and, if necessary, migrates to it (step 3212 - Figure 32B). It then sets its available bit (step 3214) and requests to join SMP operation on the new partition (step 3216). This partition remains the "permanent owner" of this CPU until a change is made in its owner field.

[0336] When an operating system instance crashes, the CPUs that are active in the partition will continue to be a part of the same instance at reboot. The CPUs do not migrate automatically to their nominal "owners". Nor do CPUs which are "owned" by a partition migrate back to an operating system instance which is crashing or rebooting. The available bit in the per-CPU flags in the HWRPB indicates the current ownership. This is also reflected in the current_owner field of the CPU node in the configuration tree.

[0337] The operating system may implement an automatic migration of secondary CPUs as part of its crash logic. That is, when a secondary CPU reaches the end of its crash logic, and would typically enter a waiting state, the operating system can implement a policy to cause the CPUs to instead migrate to a pre-defined partition. This would allow implementation of directed warm failover systems where the CPUs immediately are available at the warm backup partition when the primary application partition fails.

To remove memory from a system, it is first determined if the range of pages is private, shared, or unowned. If the memory is already marked unowned, it is considered to be removed.

[0345] For the removal of private memory, a removed page list of page frame numbers, similar to a free page list, is established to support the removal of memory. If it is determined at any time that the pages cannot be removed on line, the system can be shut-down, the pages configured as unowned with console software, and the operating system instance restarted. If the console is using some or all of the pages in the range, the console relocates itself to a different set of pages. This relocation may be accomplished in a variety of ways. For example, the console may copy its pages to another sort of pages owned by the partition, then start executing on the other set of pages. Alternatively, another console within another partition can dissolve the partition, reassign the range of pages to "unowned", then reform the partition without the memory. Additionally, the entire system could be shut down, the memory removed, and the system restarted. The removal operation can be abandoned at anytime by moving the pages from the removed page list to the free, zeroed, or bad page lists.

[0346] If the range of pages to be removed is in private memory, the following steps are repeated until all pages can be removed or the removal operation is abandoned. The process is illustrated in the flowchart of Figure 33A-33B. Before entering the process illustrated in the flowchart of Figure 33A-33B, it is determined whether there is sufficient memory to allow the removal of the memory. A system parameter, fluid page count, is typically employed to indicate the amount of spare memory readily available. If this fluid page count is too low, that is if there are insufficient fluid pages in the system to accommodate the removal, an error is returned to the user, otherwise, each page in the range to be removed is examined as described in the steps illustrated in Figure 33A-33B.

[0347] The process begins in step 3300 and proceeds from there to step 3302 where the first unremoved page within a page range to be removed is located. From step 3302, the process proceeds to step 904, where it is determined whether the page or pages to be removed have outstanding input/output operations and these pages are skipped; once their I/O operations are completed, the pages can be reconsidered for removal. If the pages have outstanding I/O, the operation returns to step 3302 where the first unremoved page is located and from there to step 3304 as described previously. To allow the system to perform other work, a system may want to stall at this point waiting for I/O to complete.

[0348] If the current unremoved page has no I/O pending, the process proceeds from step 3304 to step 3306, where it is determined whether the page is on the free, zeroed, untested, or bad page lists. If any of the memory pages are on any of these lists, the page is removed from the list in step 3308 and placed on the removed page list. Free pages are handled first so that copied pages are not copied onto free pages that are also part of the reconfigured range. From step 3308, the process proceeds to step 3326 where it is determined whether all pages have been placed in the removed page list and, if they have, the process proceeds to finish in step 3328. On the other hand, if all pages have not been placed on the removed page list, the process returns to step 3302 and from there as previously described. If desired, an implementation may choose to limit the number of iterations and execute an error recovery mechanism to avoid looping forever.

[0349] If there are no pages on the free, zeroed, untested, or bad page lists, the process proceeds from step 3306 to step 3310 where it is determined whether there are any pages on the modified page list. If there are no pages on the modified page list, the process proceeds to step 3314. On the other hand, if there are pages on the modified page list, the process proceeds to step 3312 where the pages are written to backing storage such as a system page file. From step 3312 the process proceeds to step 3326 and from there as previously described. On the other hand, if the process had proceeded to step 3314 from step 3310, in step 3314, it is determined whether any pages are process pages, i.e, pages where an application runs. Since each process may have multiple threads, each process has its own page table apart from other processes. If any pages are process pages, an AST which provides a mechanism for executing within the process' context to gain access to the process page tables, is sent to the process and the AST sets the process single threaded if necessary to synchronize access to the page tables. New page frame numbers are allocated for the pages and the contents of the pages are copied to the new page frame numbers. The old page frame numbers are placed on the instance's removed page list in step 3316. From step 3316, the process proceeds to step 3326 and from there as described previously.

[0350] If it is determined in step 3314 that none of the pages are process pages, the process proceeds to step 3318 where it is determined whether any of the pages are part of a global section i.e., a set of private pages acesssed by several processes simultaneously. If any of the pages are part of global section, the process proceeds from step 3318 to step 3320. In step 3320, the name of the global section may be displayed to a user so the user can determine which application to shut down in order to free memory. Alternatively, an operating system which can track where pages are mapped, could suspend all processes mapped to the section, copy all pages, modify all process page table entries, and place the old page frame numbers on the removed page list. From step 3320, the process would then proceed to step 3326 and from there as described previously. If in step 3318 it is determined that there are no pages which are part of the global section, the process proceeds to step 3322 where it is determined whether any page is mapped into system address space. If none of the pages is mapped into the system address space, the process proceeds from step 3322 to step 3326 and from there as previously described.

would allow a user to assist in shutting down the application which is using the shared memory. From step 3420, the process would then proceed to step 3426 and from there as previously described.

[0358] If, in step 3418, it is determined that none of the pages are part of a shared memory region that is mapped into system space, the process proceeds to step 3422 where it is determined whether any pages are part of a shared memory global section, i.e., a set of shared page mappings into one or more process address space so all the processes can have access to the same pages. If there are some pages that are part of a shared memory global section, the process proceeds to step 3424, where a callback routine is called on all instances that have this global section mapped. All processes that are mapped to the global section can be suspended, the code can then copy all data from one set of pages to another, modify all process page table entries, modify the global section data structures and place the old page frame numbers on the common property partition removed page list. Alternatively, each process that maps to the global section can be notified to shut down, or, the names of the global sections can be displayed so the user can determine which application should be shut down to assist in the removal operation. From step 3424, the process would then proceed to step 3426 and from there as previously described.

[0359] To ensure, if I/O has been initiated to shared memory, that the memory is not reused for another purpose before we are sure that all I/O in shared memory is completed, an I/O device may interrupt the instance when I/O has completed, the system would then record that the I/O is complete. Alternatively, when all I/O buses within the instance's partition have been reset, I/O has been completed. In the SHM_DESC structure in private memory, there is a field called I/O refcnt, which stands for I/O reference count. There is one SHM_DESC structure per shared memory region.

[0360] In the SHM_REG structure in the APMP database, there is a bitmask called the "attached bitmask." There is one SHM_REG structure per shared memory region. The attached bitmask contains one bit for each instance in the APMP system. If a bit in the attached bitmask is set, the corresponding instance is attached to the region.

[0361] In the configuration tree structure, maintained by the console, there is an ID field in the partition node that can be set or cleared by calling a console callback routine. During system boot after the partition's I/O buses have been reset this field in the instance's partition node is cleared.

[0362] When a sharing set is being created:

1) Loop through all other instance's partition nodes in this community.
2) If the ID field is non-zero and there is no instance running on the partition as indicated within the configuration tree, reset all I/O buses connected to the partition.
3) Set the ID field in this instance's partition node to the ID (a number that increases each time the APMP database is recreated).

[0363] When an instance is joining a sharing set:

1) Set the ID field in the instance's partition node to the current ID.

[0364] During a sharing set exit:

1) Call a routine which detaches from all shared memory regions.
2) Clear the ID field in the instance's partition node in the config tree.

[0365] When an I/O is initiated, the routine shm_reg_incref is called for each page on which I/O will be performed. When the I/O is completed, the routine shm_reg_decref is called for each page.

Routine shm_reg_incref:

Input: Address of PFN database entry for page

Read the shared memory region id from the PFN database entry.
Obtain the SHM_DESC address in private memory for this region.
Increment I/O refcnt.
Routine shm_reg_decref:

Input: Address of PFN database entry for page

Read the shared memory region id from the PFN database entry.
Obtain the SHM_DESC address in private memory for this region.
Decrement I/O refcnt

The routine shmem_APMPDB_recover is called on at least one of the other instances. In this case, the ID Field in the instance's partition node is not cleared.
Routine shmem_APMPDB_recover:

Input: Instance ID of the dead node

Loop through all SHM_REG structures in the APMP database.
If the crashing instance was attached to the region:

1) If the ID field in the instance's partition node is non-zero, this might indicate that the instance was halted, not crashed. I/O devices may still be writing to shared memory.

Loop to the next SHM_REG structure leaving this instance attached and the shared memory region in place. When the instance joins the sharing set again, it will clear the attached bit after having reset the I/O buses. It will delete the shared memory region if it was the last instance attached to the region.

2) If the ID field in the instance's partition node is clear:

clear the bit for the instance in the attached bitmask

If the attached bitmask has no bits set:

Call shm_reg_delete

Loop to the next SHM_REG structure
After all SHM_REG structures have been processed, execute more shared memory recovery code.

[0368] Routine shmem_sharing_set_join:

Input: Instance ID of this instance.
Map to the APMP database in shared memory.
Execute other shared memory community join code.
Loop through all SHM_REG structures in the APMP database.
If this instance was attached to the region:

- Clear the bit for this instance in the attached bitmask

If the attached bitmask has no more bits set

- Call shm_reg_delete

Loop to the next SHM_REG structure
After all SHM_REG structures have been processed, return.

[0369] To choose the initial APMP database pages, the routine shmem_config_APMPDB is called by APMPDB_map_initial to choose the intial set of APMPDB pages.

Data structures:

[0370] The community node in the config tree contains a 64-bit field, called APMPDB_INFO, which is used to store APMPDB page information. The first 32-bits, APMPDB_INFO[31:0], is the low PFNof the APMPDB pages. The second 32-bits, APMPDB_INFO[63:32], is the page count of APMPDB pages.

[0371] Each instance keeps an array in private memory called the "shared memory array." Each element in the array contains a shared memory PFN and a page count. The entire array describes all shared memory owned by the community that this instance is a part of.

[0372] The configuration tree may contain tested memory bitmaps for shared memory. If the configuration tree does not contain a bitmap for a range of memory, the memory has been tested and it is good. If a bitmap exists for a range of memory, each bit in the bitmap indicates whether a page of shared memory is good or bad.

Call SET_APMPDB_INFO to write PFN pages intothe APMPDB field
If SET_APMPDB_INFO returns an error, return to (2)
If SET_APMPDB_INFO returns success, return PFN and
PAGES to the caller

(6.4) If a bad page is encountered within the range

Set PFN to the highest numbered bad PFN+1
If PFN+PAGES-1 is still within the shared memory array element return to (6.2)
If PFN+PAGES-1 is greater than the range described by this shared memory array element, move to the next shared memory array element

(6.5) If there are no shared memory array elements left, return an error.

[0376] An illustrative page frame number database layout is illustrated by the memory map of Figure 11. In this illustrative example the system includes two instances, A and B, each with 64 megabytes of private memory and 64 megabytes of shared memory. The memory is arranged as eight kilobyte pages, with private memory for instance A extending from page frame number (PFN) 0 through PFN 1 BFF (hexadecimal). Sixty four megabytes of shared memory extends from PFN 2000 to PFN 3FFF. Private memory for instance B extends from PFN 800000 through PFN 801FFF. The memory used to hold the PFN database for instance A comes from instance A's private memory (0-1FFF), the memory used to hold the PFN database for instance B comes from instance B's private memory (2000-3FFF), and the memory used to hold the PFN database for shared memory comes from the shared memory (800000-801FFF). Instance A cannot access the PFN database entries for Instance B's memory because, as illustrated, that memory region is not mapped into the system space for instance A. Similarly, Instance B cannot access the PFN database entries for Instance A's memory because that memory region is not mapped into the system space for instance B. Both Instances A and B map the shared pages into the PFN database entries for shared memory. Instances map to the shared pages with private memory page tables because private memory and shared memory PFN database entries may be mapped by the same page table pages. As noted above, the granularity of physical memory may be chosen as the least common multiple of PFN database entry size and memory pages size. In the illustrative example the memory page size is 8 kilobytes and the granularity of physical memory is equal to the page size squared divided by eight (bytes), or 8 MB. Page sizes of 16, 32, and 64 KB yield physical memory granularity of 32, 128, and 512 MB, respectively.

L. RECOVERING SHARED MEMORY REGIONS IN A MULTI-PROCESSOR SYSTEM (FIGS. 19-21)

[0377] In accordance with a further aspect of the present invention, multiple instances of operating systems execute cooperatively in a single multiprocessor computer wherein all processors and resources are electrically connected together. The single physical machine with multiple physical processors and resources is adaptively subdivided by software into multiple partitions, each with the ability to run a distinct copy, or instance, of an operating system. Each of the partitions has access to its own physical resources plus resources designated as shared. In accordance with one embodiment, the partitioning of resources is performed by assigning resources within a configuration.

[0378] More particularly, software logically, and adaptively, partitions CPUs, memory, and I/O ports by assigning them together. An instance of an operating system may then be loaded on a partition. At different times, different operating system instances may be loaded on a given partition. This partitioning, which a system manager directs, is a software function; no hardware boundaries are required. Each individual instance has the resources it needs to execute independently. Resources, such as CPUs and memory, can be dynamically assigned to different partitions and used by instances of operating systems running within the machine by modifying the configuration. The partitions themselves can also be changed without rebooting the system by modifying the configuration tree. The resulting adaptively partitioned, multi-processing (APMP) system exhibits scalability, flexibility, and high performance.

[0379] Memory may be reconfigured into or out of a partition or community under software control and hardware hot in-swapping or out-swapping are supported. In general, memory may be in one of three states: private, shared, or unowned. Memory is private if it is "owned" by a single system partition. Memory is shared if it is owned by a community. A community, or sharing set, is a collection of one or more partitions which may share resources. Otherwise, it is unowned. Memory may be reconfigured between any of three states directly. For example, memory may be reconfigured from private in one partition to private in another partition, or from shared in a community to private in a partition. Memory is placed in the unowned state by an operating system instance and console software, reflected in the system configuration tree, before the memory can be out-swapped or in-swapped. A page frame number database is sized to include all possible memory that can be in-swapped and added memory pages may be employed as page frame

a console program which maintains configuration information indicating which of the plurality of processors, which memory portions and which I/O circuitry portions are assigned to each partition; and wherein the software mechanism for assigning system resources comprises a first data storage for maintaining configuration information indicating which of the plurality of processors is assigned to each partition; and a mechanism for modifying the configuration information to change an assignment of assignable system resources from one partition to another.

4. A computer system according to claim 1 further characterized by a migration mechanism for migrating assignable system resources from one partition to another during system operation without a reboot of the entire system.

5. A computer system according to claim 4 wherein the migration mechanism comprises a first communication mechanism which communicates with an operating system instance in a partition from which a processor is being moved to cause the operating system instance to quiesce the processor; a second communication mechanism which communicates with an operating system instance in a partition to which a processor is being moved to cause the operating system instance to begin using the processor; a third communication mechanism which communicates with an operating system instance in a partition from which memory is being moved to cause the operating system instance to unload the memory; a fourth communication mechanism which communicates with a plurality of operating system instances which share memory to be moved to cause the operating system instances to cooperatively unload the memory; and a fifth communication mechanism which communicates with an operating system instance in a partition to which memory is being moved to cause the operating system instance to begin filling the memory.

6. A computer system according to claim 1, further characterized by a first pair of operating system instances sharing a first subset of the plurality of system resources and a second pair of operating system instances sharing a second subset of the plurality of system resources different from the first subset of operating system resources.

7. A computer system according to claim 6 wherein some memory being dedicated to each operating system instance in the first pair of operating system instances; some memory being dedicated to each operating system instance in the second pair of operating system instances; and some of the memory being shared among the first pair of operating system instances and the second pair of operating system instances.

8. A computer system according to claim 7 wherein the shared part of memory comprises a shared cache-coherent memory.

9. A computer system according to claim 1, further characterized by:

a processor designation storage device that stores a designation indicative of the operational status of a first processor relative to a first instance, the designation being used by the first instance in determining whether the first processor is currently available to it for processing activities.

10. A computer system according to claim 9 wherein the designation comprises a memory bit associated with the first instance, a first particular setting of the first memory bit indicating that the processor is under the control of the first instance.

11. A computer system according to claim 9 wherein the processor designation storage device stores (A) designations for each of a plurality of processors to define the association of each processor with one of said plurality of instances, the designations being used by the one instance to determine whether a given processor is currently available to it for processing; (B) a set of designations for each instance, each set of designations being associated with and examinable by its respective instance and indicating which, if any, of the processors is under the control of its respective instance, and whether a corresponding one or more of said processors has a particular operational status represented by that set.

12. A computer system according to claim 1, further characterized by:

at least one group of partitions forming a community that shares memory.

13 A computer system according to claim 12 wherein a segment of the shared memory forms a database for the community's shared memory for indicating whether an operating system instance associated with a partition within the community is active.

indication of availability comprises a flag that is set by the first processor.

26. A computer system according to claim 1, further characterized by;

a processor migration apparatus that reassigns a first processor from a first partition to a second partition without a reboot of the entire system, wherein said migration apparatus stores a processing context of the processor relative to the first partition prior to the reassignment, and, after the reassignment of the first processor, the first processor loads any processing context that it may have stored during a previous execution with the second partition.

27. A computer system according to claim 26 further comprising a set of hardware flags associated with each partition, the hardware flags for a particular partition including at least one flag indicating the operational status of a particular processor executing on that partition, and an availability flag indicating whether said particular processor is available to join symmetric multiprocessing (SMP) on that partition.

28. A computer system according to claim 27 wherein each said set of hardware flags include an ownership flag, and, prior to the reassignment of the first processor, a first one of the ownership flags for the first partition indicates that the first processor is under the control of an instance running on the first partition, while, after the reassignment of the first processor, the first ownership flag indicates that the first processor is no longer under the control of the instance running on the first partition and is instead under the control of an instance running on the second partition.

29. A computer system according to claim 1, further characterized by:

a processor migration apparatus that reassigns a first processor from a first partition to a second partition, while it is executing under the control of an operating system instance running on the first partition, provides an indication of the destination partition to the first processor, and instructs the first processor to cease execution under the control of the instance running on the first partition.

30. A computer system according to claim 29 wherein the plurality of processors is divided into groups and wherein each group comprises a console program to which control of the processor is transferred prior to its reassignment to the second partition, and wherein said migration apparatus causes the first processor to enter an unassigned state prior to its reassignment to the second partition.

31. A computer system according to claim 1, further characterized by:

an inter-instance communication apparatus that enables communication between a first instance running on a first partition and a second instance running on a second partition by modification of shared memory to which both instances have access.

32. A computer system according to claim 31 wherein the communication apparatus can effect a given communication by modifying only one bit of the shared memory, wherein the bit that is modified is part of a notification bitvector of the second instance, the notification bitvector comprising a plurality of bits, each of which corresponds to a different predetermined event, and can have its value changed by the first instance to indicate the occurrence of the event to the second instance; and wherein the second instance uses the position of the bit in the bitvector as an index to locate in memory an appropriate task to initiate in response to the modification of that bit.

33. A computer system according to claim 32 wherein the communication apparatus includes apparatus under the control of the first instance that causes the generation of an interrupt from the first instance to the second instance to indicate to the second instance that said bit has been modified.

34. A computer system according to claim 31 wherein the communication apparatus further comprises a packet sending apparatus that stores in shared memory packetized data to be communicated from the first instance to the second instance.

35. A computer system according to claim 34 wherein the communication apparatus, for a given communication, modifies one notification bit in the shared memory, the modification of the bit indicating to the second instance that the packet has been stored; and said system further comprises apparatus for causing the generation of an interrupt to the second instance to indicate to the second instance that the notification bit has been modified; wherein, upon

(c) forming a community of partitions by sharing memory among a group of partitions; and
(d) forming a shared memory database for the community within the shared memory, including within the shared memory database an indication of whether an operating system instance associated with the community is active, each instance mapping to the shared memory database as it joins a community.

44. A method according to claim 43 wherein step (d) comprises the step of:

locking the shared memory database to block access to the database by another instance as an instance examines the shared memory database.

44. A method according to claim 39, further characterized by the steps of:

(c) forming a community of partitions by sharing memory among a group of partitions; and
(d) placing memory into an unowned state, whereby the memory is unowned by any partition, whenever the memory is to be added to or deleted from the system.

45. A method according to claim 39, further characterized by the steps of:

storing an indication of a destination instance to which control of a first processing resource of the first instance is to be transferred upon a failure within the first instance;
determining, upon the occurrence of said failure, the destination instance for the first processing resource; and changing the indicia of control for the first processing resource to transfer control of the first processing resource from the first instance to the destination instance without operator involvement.

46. A method according to claim 39, further characterized by the step of:

providing an assignment of permanent ownership of each processor to a respective partition, the assignments being used to associate the processors with their respective partitions during initialization of the system, including the step of storing information regarding the assignments in non-volatile memory.

48. A method according to claim 39, further characterized by the step of:

reassigning a first processor from a first partition to a second partition without rebooting the entire system, wherein said reassigning comprises causing, prior to the reassignment, the first processor to store a processing context relative to the first partition, and causing the first processor, after reassignment, to load any processing context which it may have stored from a previous execution with the second partition.

49. A method according to claim 39, further characterized by the step of:

reassigning a first processor from a first partition to a second partition, wherein said reassigning comprises causing the processor to terminate a first processing state prior to its reassignment under the control of an instance running on the first partition.

50. A method according to claim 39, further characterized by the step of:

providing inter-instance communication that allows a first instance running on a first partition to communicate with a second instance running on a second partition by modifying shared memory to which both instances have access.

51. A method according to claim 50 wherein, the communication providing step includes the steps of modifying a part of a notification bitvector of the second instance, the notification bitvector comprising a plurality of bits, each of which corresponds to a different predetermined event and can have its value changed to indicate the occurrence of that event; and using the position of the bit in the bitvector as an index to locate in memory an appropriate task to initiate in response to the modification of that bit; and storing in shared memory packetized data to be communicated from the first instance to the second instance.

52. A method according to claim 39, further characterized by the steps of:

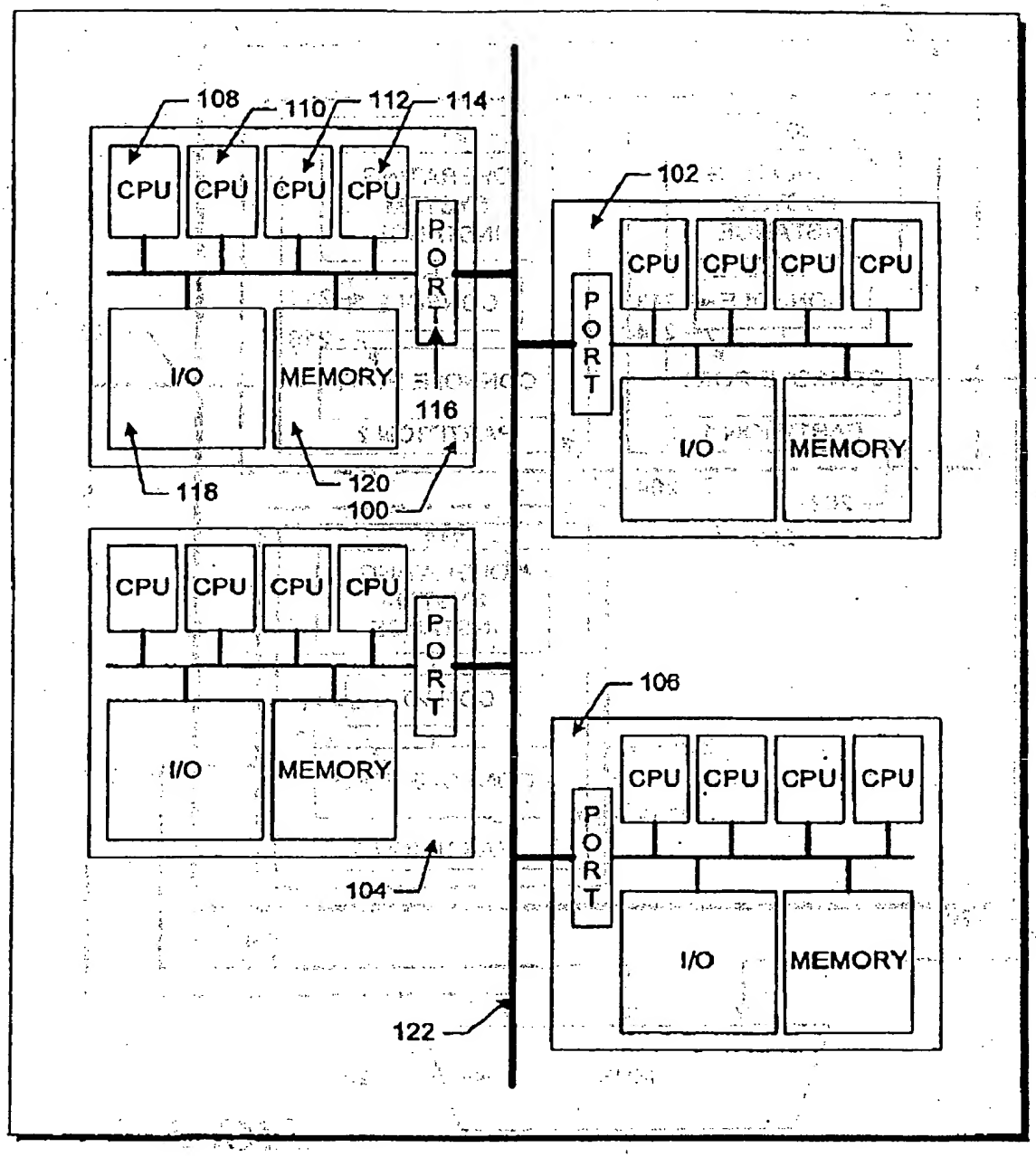

108
110
112
114
CPU
CPU
CPU
CPU
PORT
I/O
MEMORY
116
118
120
100
102
CPU
CPU
CPU
CPU
PORT
I/O
MEMORY
CPU
CPU
CPU
CPU
PORT
I/O
MEMORY
104
106
CPU
CPU
CPU
CPU
PORT
I/O
MEMORY
122

FIG. 1

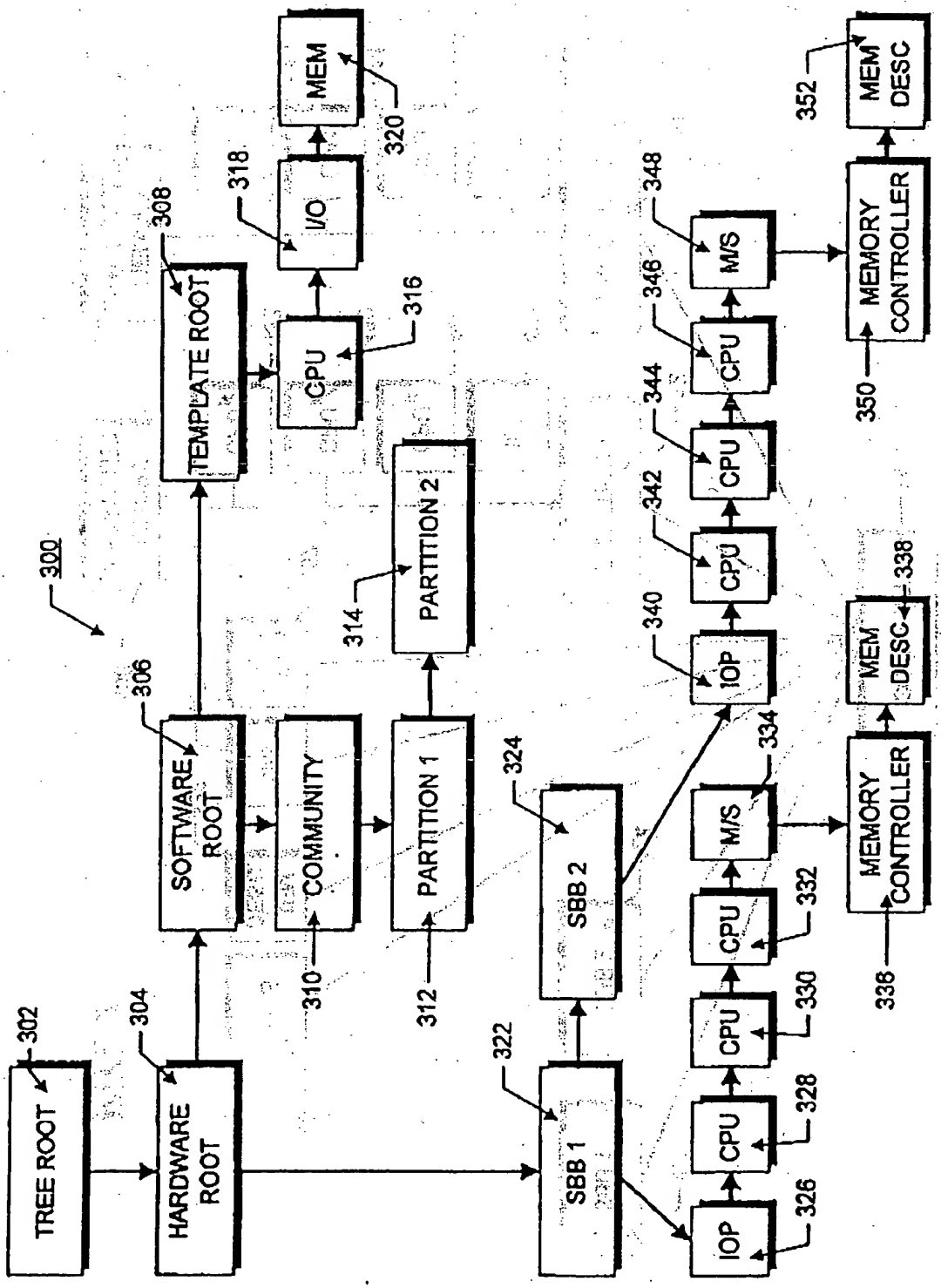

300
TREE ROOT
302
HARDWARE ROOT
304
SOFTWARE ROOT
306
TEMPLATE ROOT
308
COMMUNITY
310
PARTITION 1
312
PARTITION 2
314
CPU
316
I/O
318
MEM
320
SBB 1
322
SBB 2
324
IOP
326
CPU
328
CPU
330
CPU
332
M/S
334
IOP
340
CPU
342
CPU
344
CPU
346
M/S
348
MEMORY CONTROLLER
350
MEM DESC
352
MEMORY CONTROLLER
338
MEM DESC
338

FIG. 3

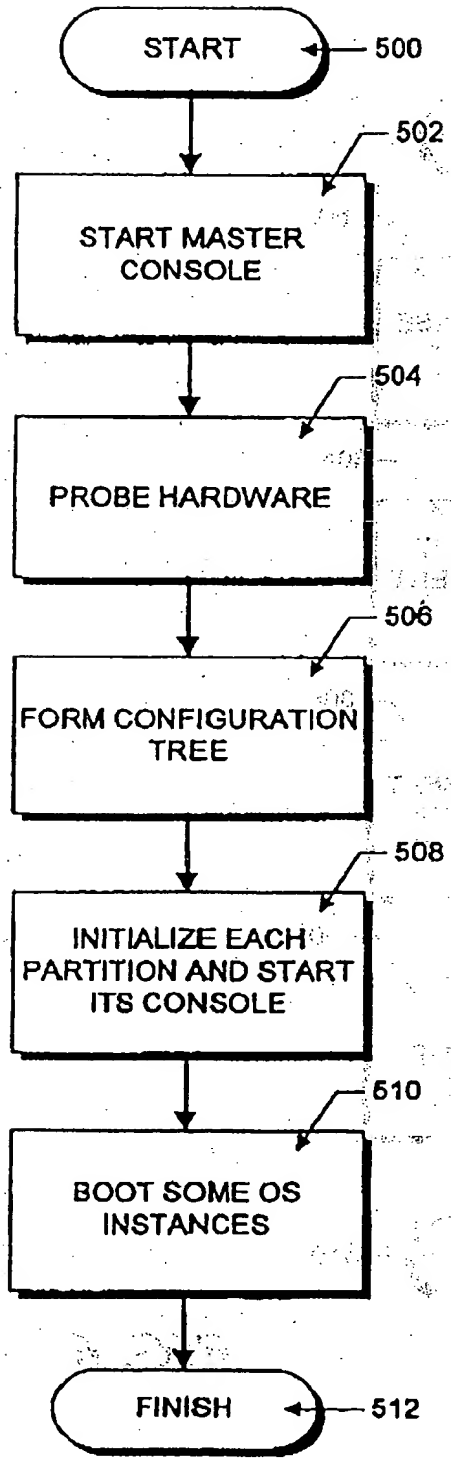

START
500
502
START MASTER CONSOLE
504
PROBE HARDWARE
506
FORM CONFIGURATION TREE
508
INITIALIZE EACH PARTITION AND START ITS CONSOLE
510
BOOT SOME OS INSTANCES
FINISH
512

FIG. 5

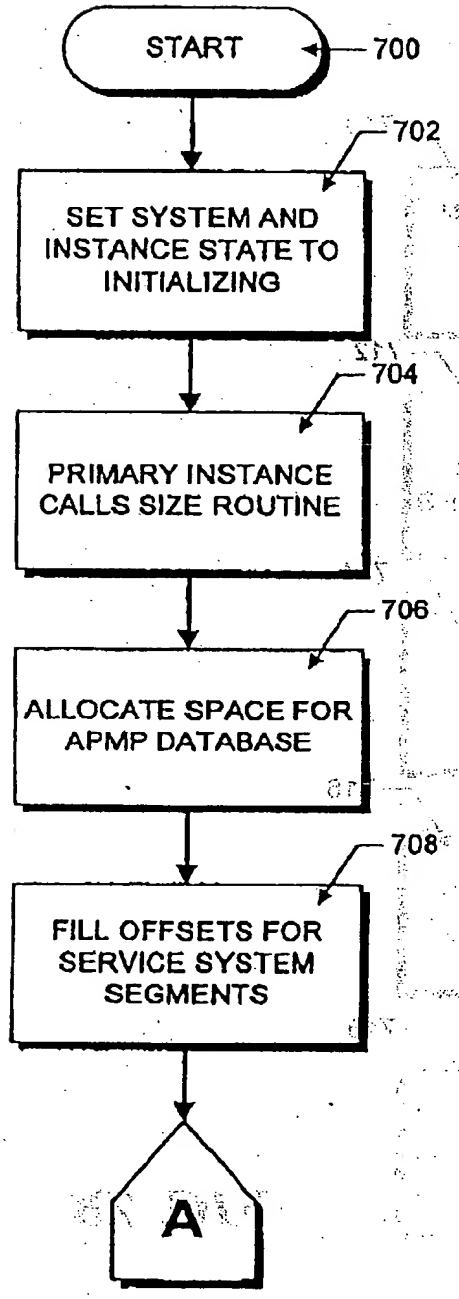

START
700
702
SET SYSTEM AND INSTANCE STATE TO INITIALIZING
704
PRIMARY INSTANCE CALLS SIZE ROUTINE
706
ALLOCATE SPACE FOR APMP DATABASE
708
FILL OFFSETS FOR SERVICE SYSTEM SEGMENTS
A

FIG. 7A

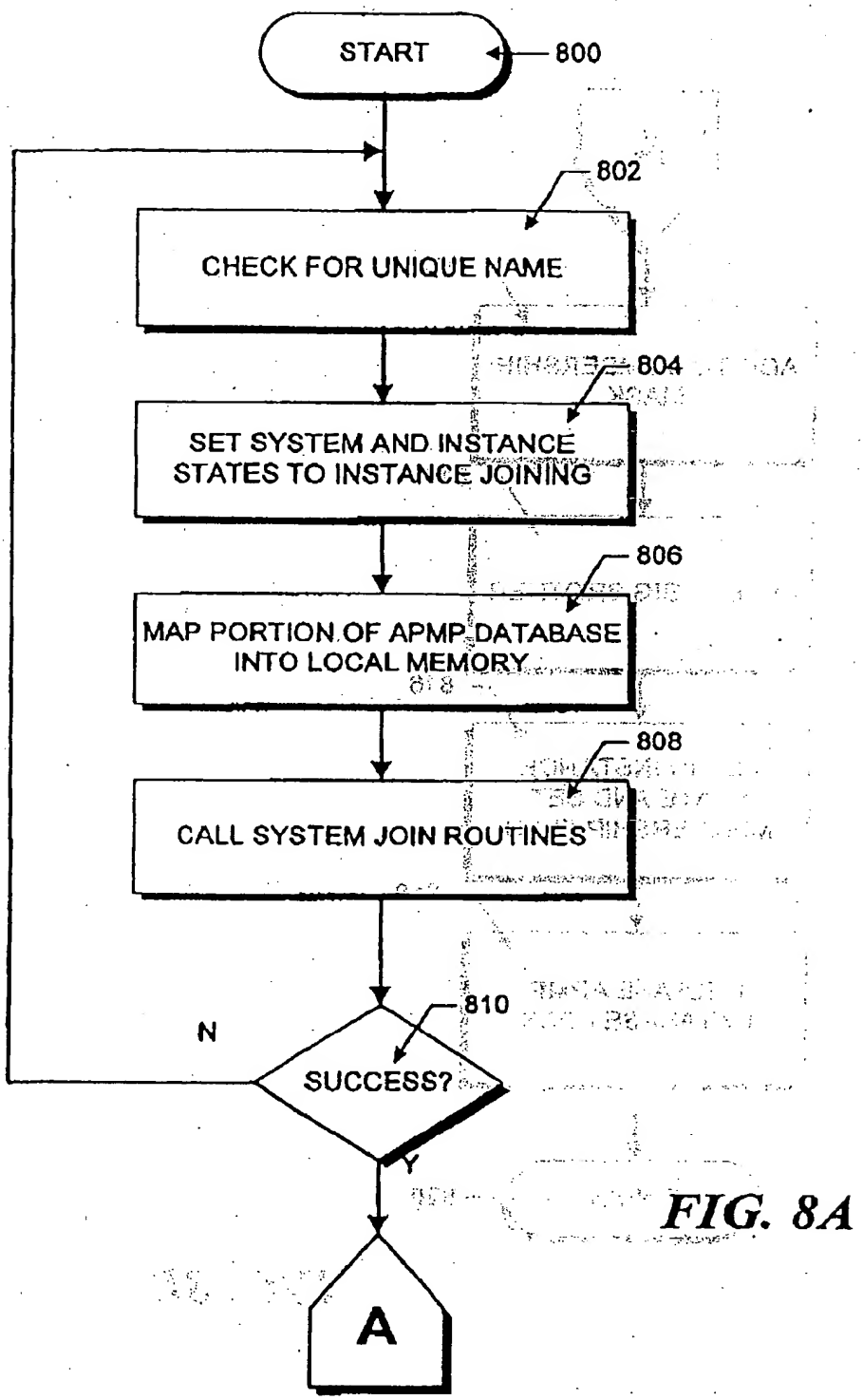

START
800
802
CHECK FOR UNIQUE NAME
804
SET SYSTEM AND INSTANCE STATES TO INSTANCE JOINING
806
MAP PORTION OF APMP DATABASE INTO LOCAL MEMORY
808
CALL SYSTEM JOIN ROUTINES
810
SUCCESS?
N
Y
A

FIG. 8A

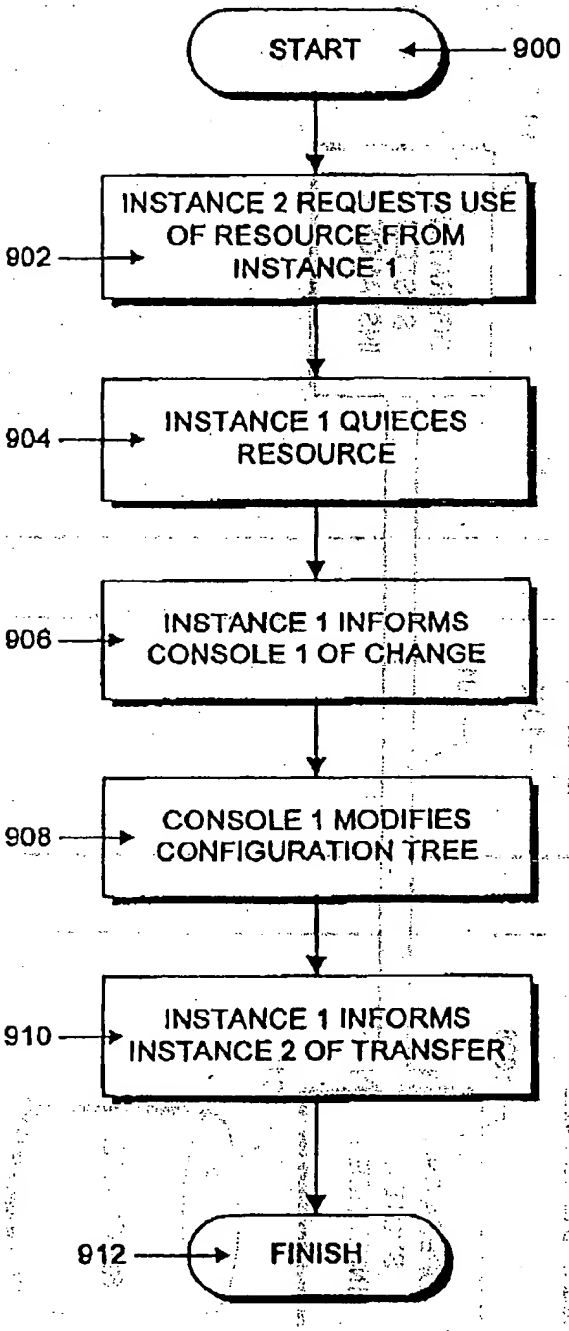

START
900
INSTANCE 2 REQUESTS USE OF RESOURCE FROM INSTANCE 1
902
INSTANCE 1 QUIECES RESOURCE
904
INSTANCE 1 INFORMS CONSOLE 1 OF CHANGE
906
CONSOLE 1 MODIFIES CONFIGURATION TREE
908
INSTANCE 1 INFORMS INSTANCE 2 OF TRANSFER
910
FINISH
912

FIG. 9

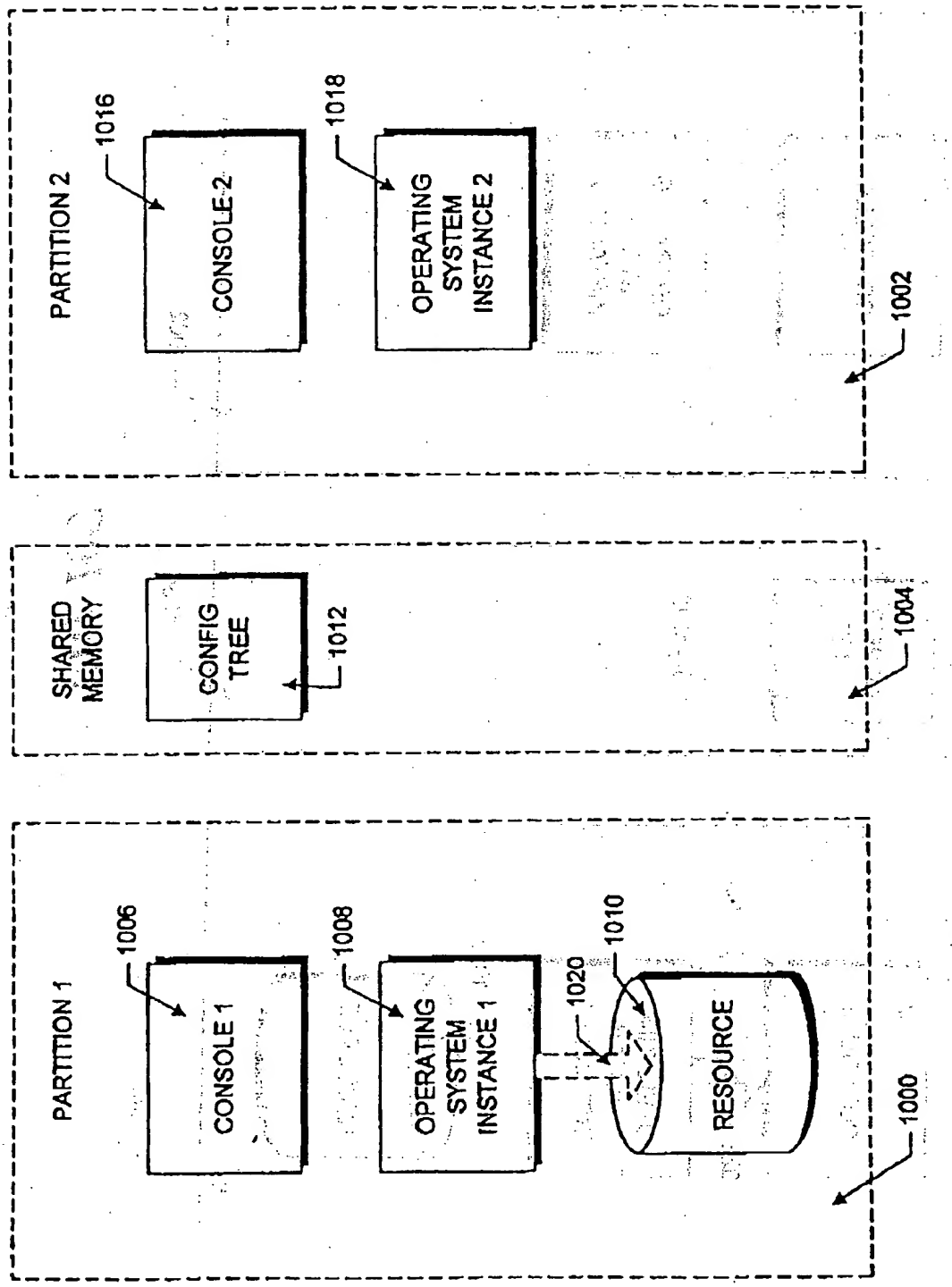

PARTITION 2
1016
CONSOLE 2
1018
OPERATING SYSTEM INSTANCE 2
1002
SHARED MEMORY
CONFIG TREE
1012
1004
PARTITION 1
1006
CONSOLE 1
1008
OPERATING SYSTEM INSTANCE 1
1020
1010
RESOURCE
1000

FIG. 10B

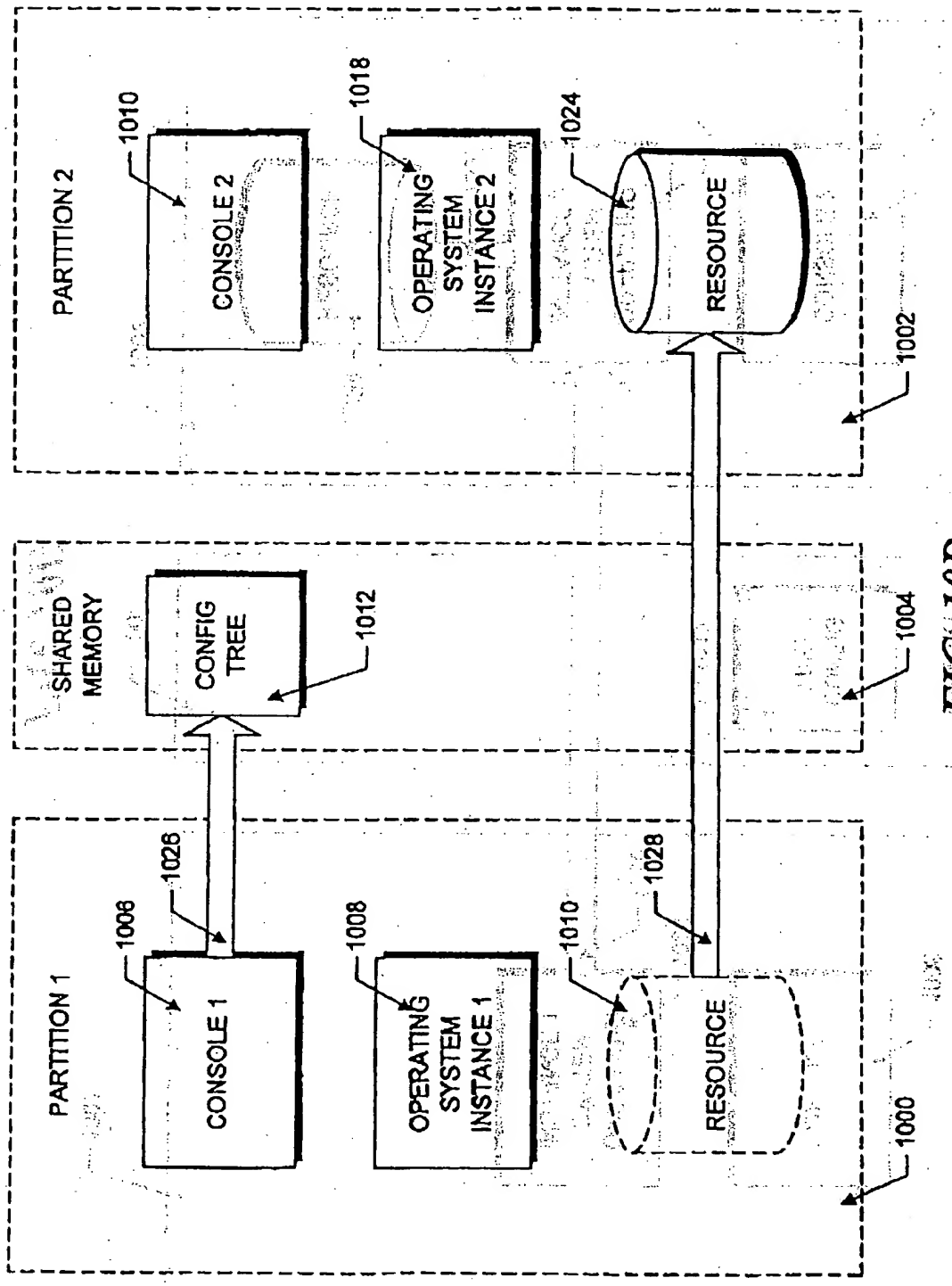

PARTITION 2
1010
CONSOLE 2
1018
OPERATING SYSTEM INSTANCE 2
1024
RESOURCE
1002
SHARED MEMORY
CONFIG TREE
1012
1004
PARTITION 1
1026
1006
CONSOLE 1
1008
OPERATING SYSTEM INSTANCE 1
1028
1010
RESOURCE
1000

FIG. 10D

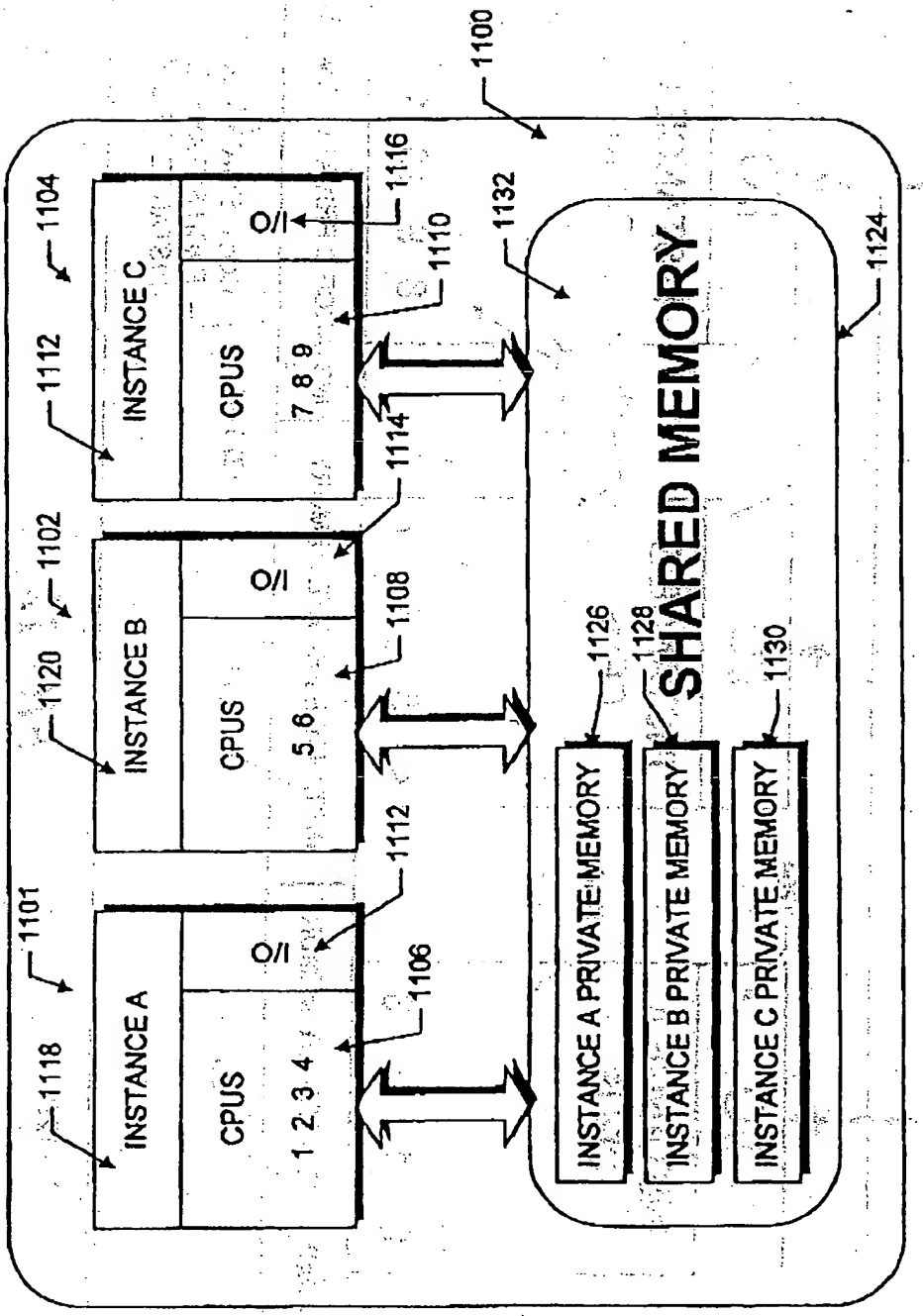

1100
1101
1102
1104
1106
1108
1110
1112
1114
1116
1118
1120
1124
1126
1128
1130
1132
INSTANCE A
CPUS
1 2 3 4
O/I
INSTANCE B
CPUS
5 6
O/I
INSTANCE C
CPUS
7 8 9
O/I
SHARED MEMORY
INSTANCE A PRIVATE MEMORY
INSTANCE B PRIVATE MEMORY
INSTANCE C PRIVATE MEMORY

FIG. 11

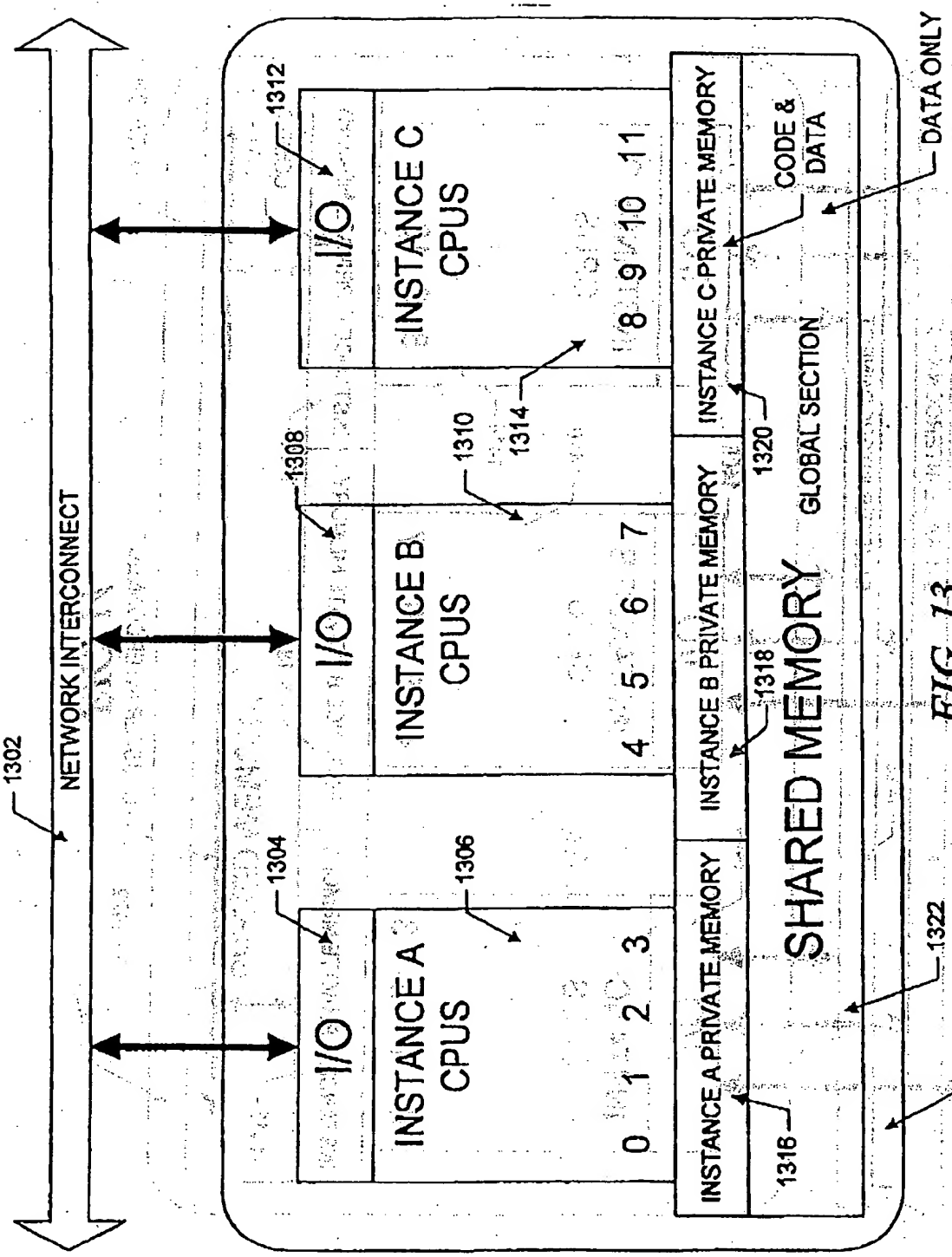

1302
NETWORK INTERCONNECT
1304
I/O
INSTANCE A CPUS
1306
0 1 2 3
1308
I/O
INSTANCE B CPUS
1310
4 5 6 7
1312
I/O
INSTANCE C CPUS
1314
8 9 10 11
INSTANCE A PRIVATE MEMORY
1316
INSTANCE B PRIVATE MEMORY
1318
INSTANCE C PRIVATE MEMORY
1320
SHARED MEMORY
GLOBAL SECTION
CODE & DATA
DATA ONLY
1322
1300

FIG. 13

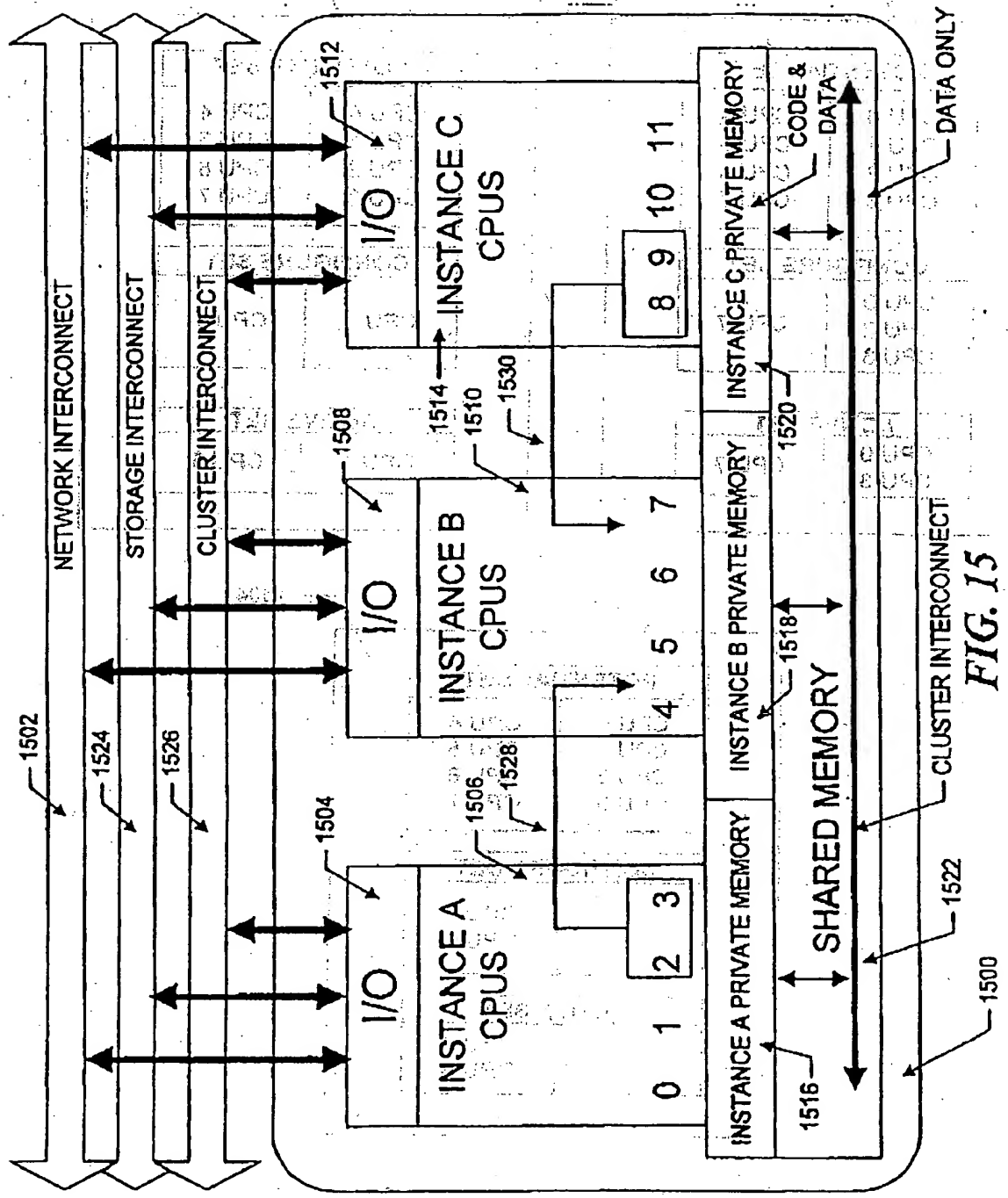

NETWORK INTERCONNECT
STORAGE INTERCONNECT
CLUSTER INTERCONNECT
I/O
INSTANCE A CPUS
INSTANCE B CPUS
INSTANCE C CPUS
0 1 2 3
4 5 6 7
8 9 10 11
INSTANCE A PRIVATE MEMORY
INSTANCE B PRIVATE MEMORY
INSTANCE C PRIVATE MEMORY
SHARED MEMORY
CODE & DATA
DATA ONLY
CLUSTER INTERCONNECT
1500
1502
1504
1506
1508
1510
1512
1514
1516
1518
1520
1522
1524
1526
1528
1530

FIG. 15

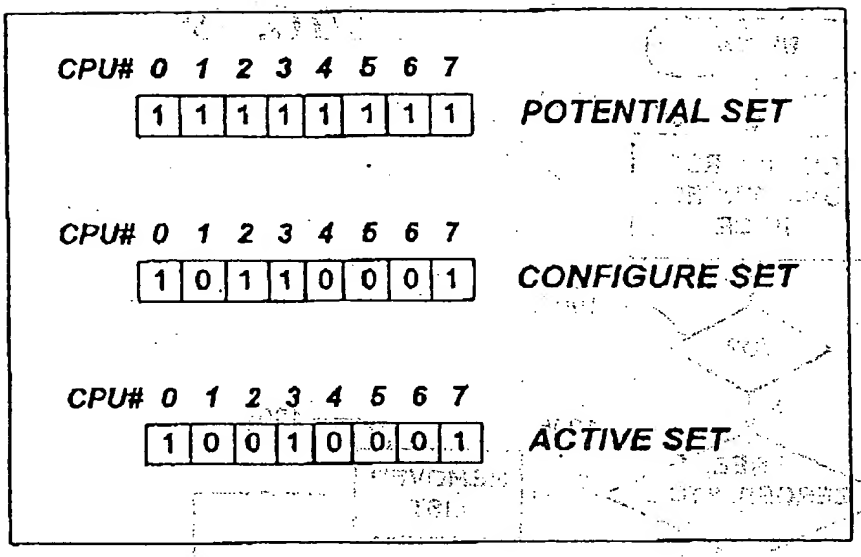

CPU# 0 1 2 3 4 5 6 7
1 1 1 1 1 1 1 1
POTENTIAL SET
CPU# 0 1 2 3 4 5 6 7
1 0 1 1 0 0 0 1
CONFIGURE SET
CPU# 0 1 2 3 4 5 6 7
1 0 0 1 0 0 0 1
ACTIVE SET

FIG. 17

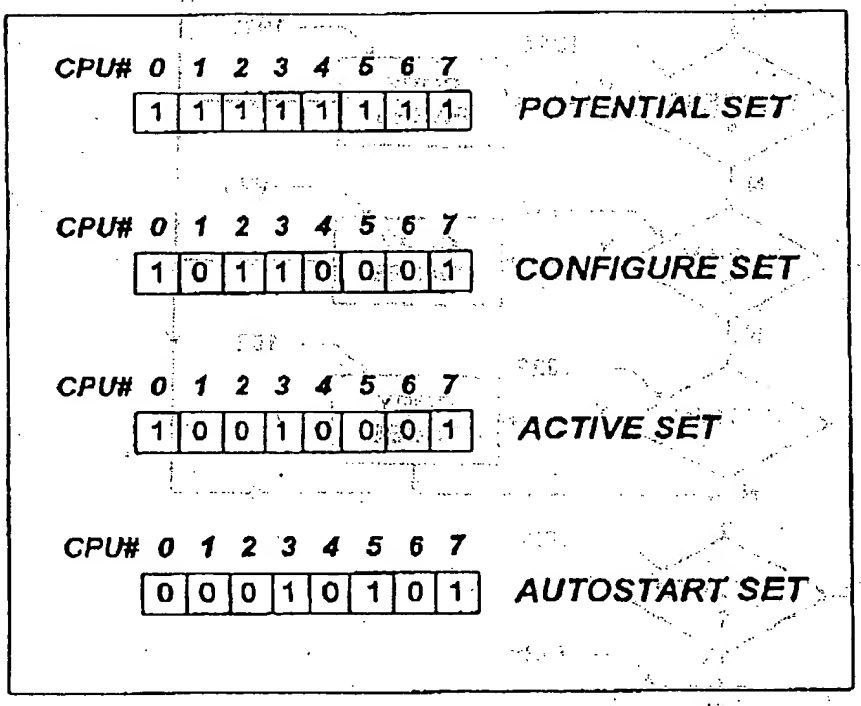
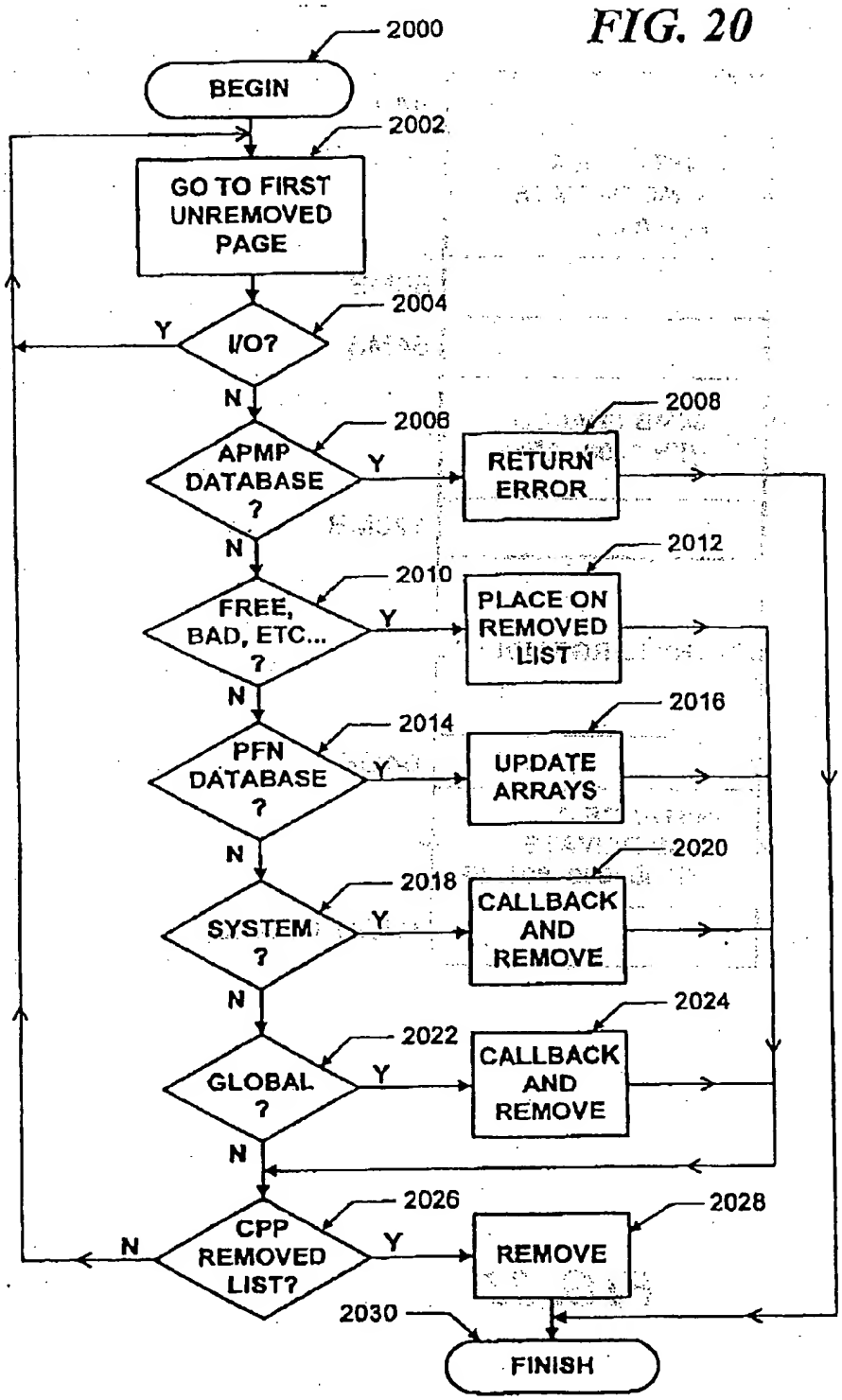

CPU# 0 1 2 3 4 5 6 7
1 1 1 1 1 1 1 1
POTENTIAL SET
CPU# 0 1 2 3 4 5 6 7
1 0 1 1 0 0 0 1
CONFIGURE SET
CPU# 0 1 2 3 4 5 6 7
1 0 0 1 0 0 0 1
ACTIVE SET
CPU# 0 1 2 3 4 5 6 7
0 0 0 1 0 1 0 1
AUTOSTART SET

FIG. 18

FIG. 20


2000
BEGIN
2002
GO TO FIRST UNREMOVED PAGE
2004
I/O?
Y
N
2006
APMP DATABASE ?
Y
2008
RETURN ERROR
N
2010
FREE, BAD, ETC... ?
Y
2012
PLACE ON REMOVED LIST
N
2014
PFN DATABASE ?
Y
2016
UPDATE ARRAYS
N
2018
SYSTEM ?
Y
2020
CALLBACK AND REMOVE
N
2022
GLOBAL ?
Y
2024
CALLBACK AND REMOVE
N
2026
CPP REMOVED LIST?
N
Y
2028
REMOVE
2030
FINISH

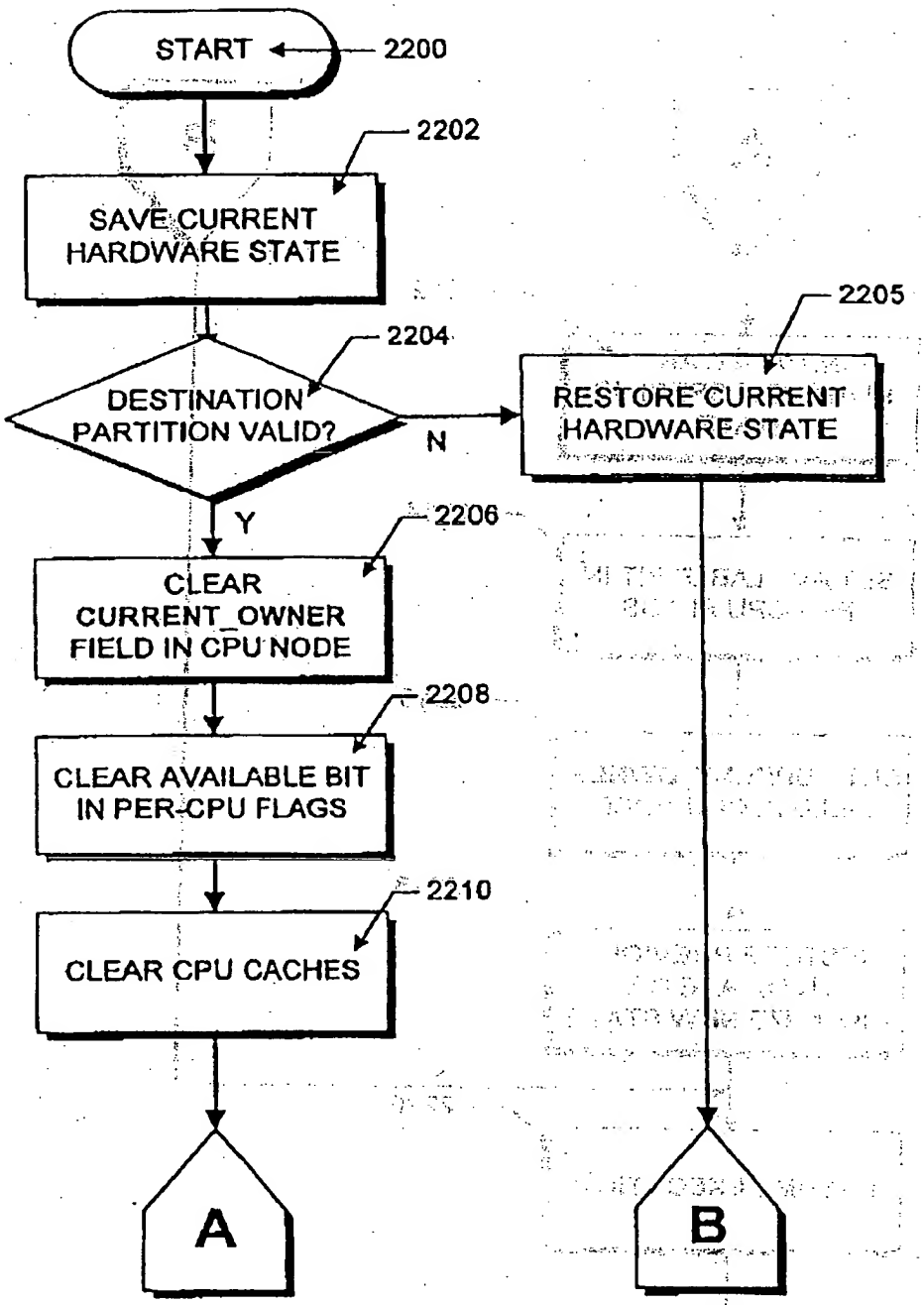

START
2200
2202
SAVE CURRENT HARDWARE STATE
2204
DESTINATION PARTITION VALID?
N
2205
RESTORE CURRENT HARDWARE STATE
Y
2206
CLEAR CURRENT_OWNER FIELD IN CPU NODE
2208
CLEAR AVAILABLE BIT IN PER-CPU FLAGS
2210
CLEAR CPU CACHES
A
B

FIG. 22A

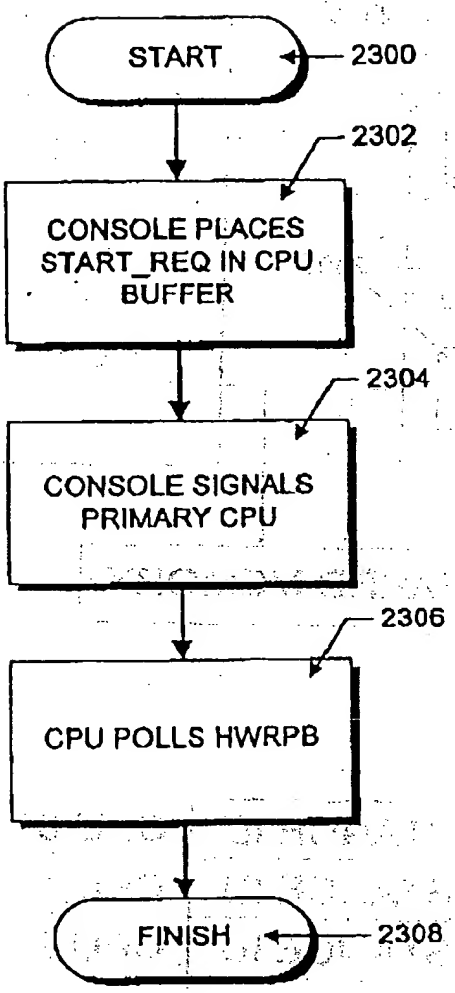

START
2300
2302
CONSOLE PLACES START_REQ IN CPU BUFFER
2304
CONSOLE SIGNALS PRIMARY CPU
2306
CPU POLLS HWRPB
FINISH
2308

FIG. 23

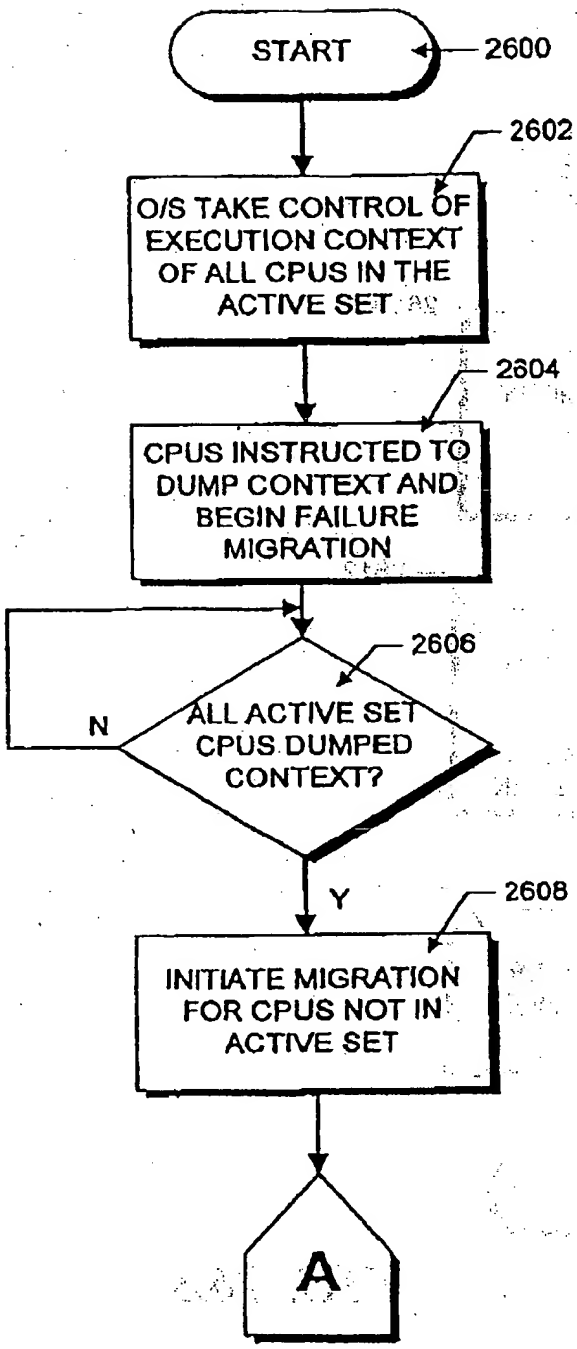

START
2600
2602
O/S TAKE CONTROL OF EXECUTION CONTEXT OF ALL CPUS IN THE ACTIVE SET
2604
CPUS INSTRUCTED TO DUMP CONTEXT AND BEGIN FAILURE MIGRATION
2606
ALL ACTIVE SET CPUS DUMPED CONTEXT?
N
Y
2608
INITIATE MIGRATION FOR CPUS NOT IN ACTIVE SET
A

FIG. 26A

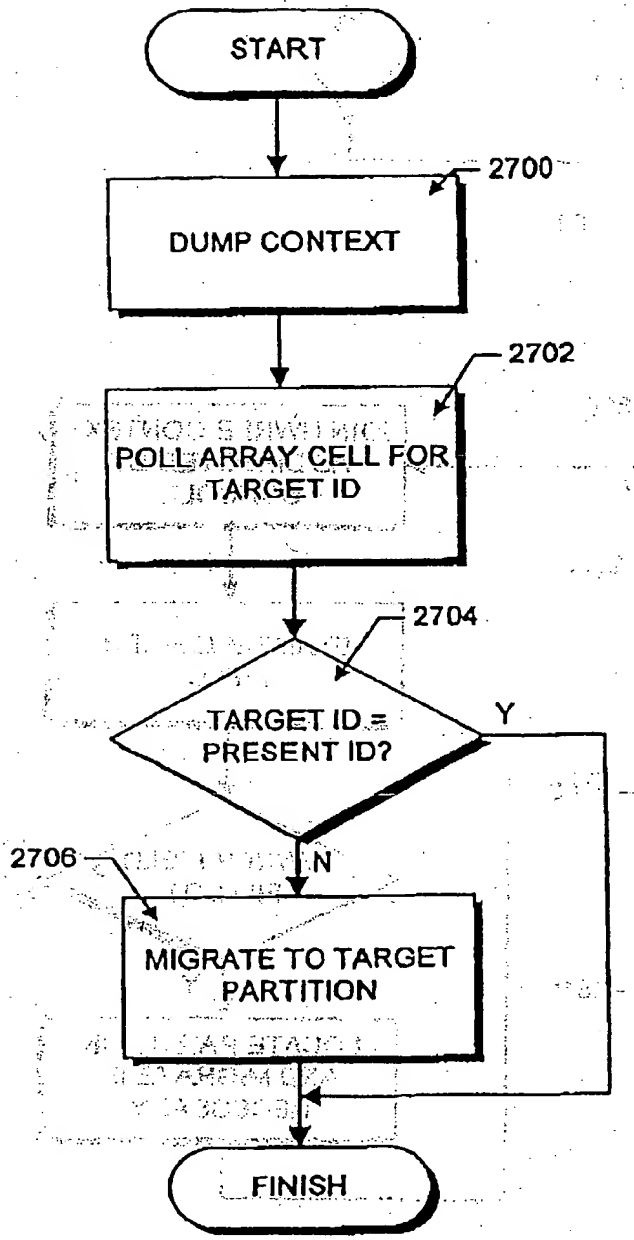

START
2700
DUMP CONTEXT
2702
POLL ARRAY CELL FOR TARGET ID
2704
TARGET ID = PRESENT ID?
Y
N
2706
MIGRATE TO TARGET PARTITION
FINISH

FIG. 27

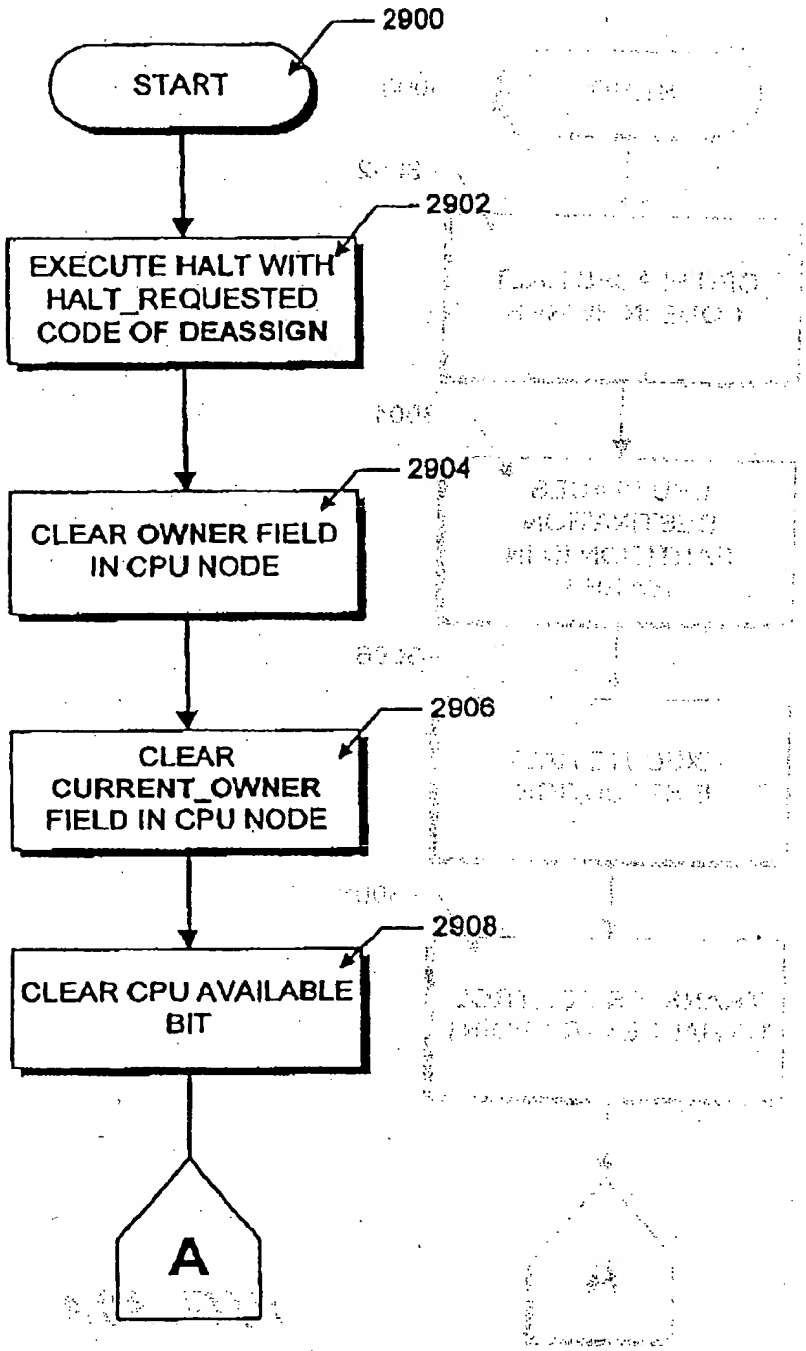

2900
START
2902
EXECUTE HALT WITH HALT_REQUESTED CODE OF DEASSIGN
2904
CLEAR OWNER FIELD IN CPU NODE
2906
CLEAR CURRENT_OWNER FIELD IN CPU NODE
2908
CLEAR CPU AVAILABLE BIT
A

FIG. 29

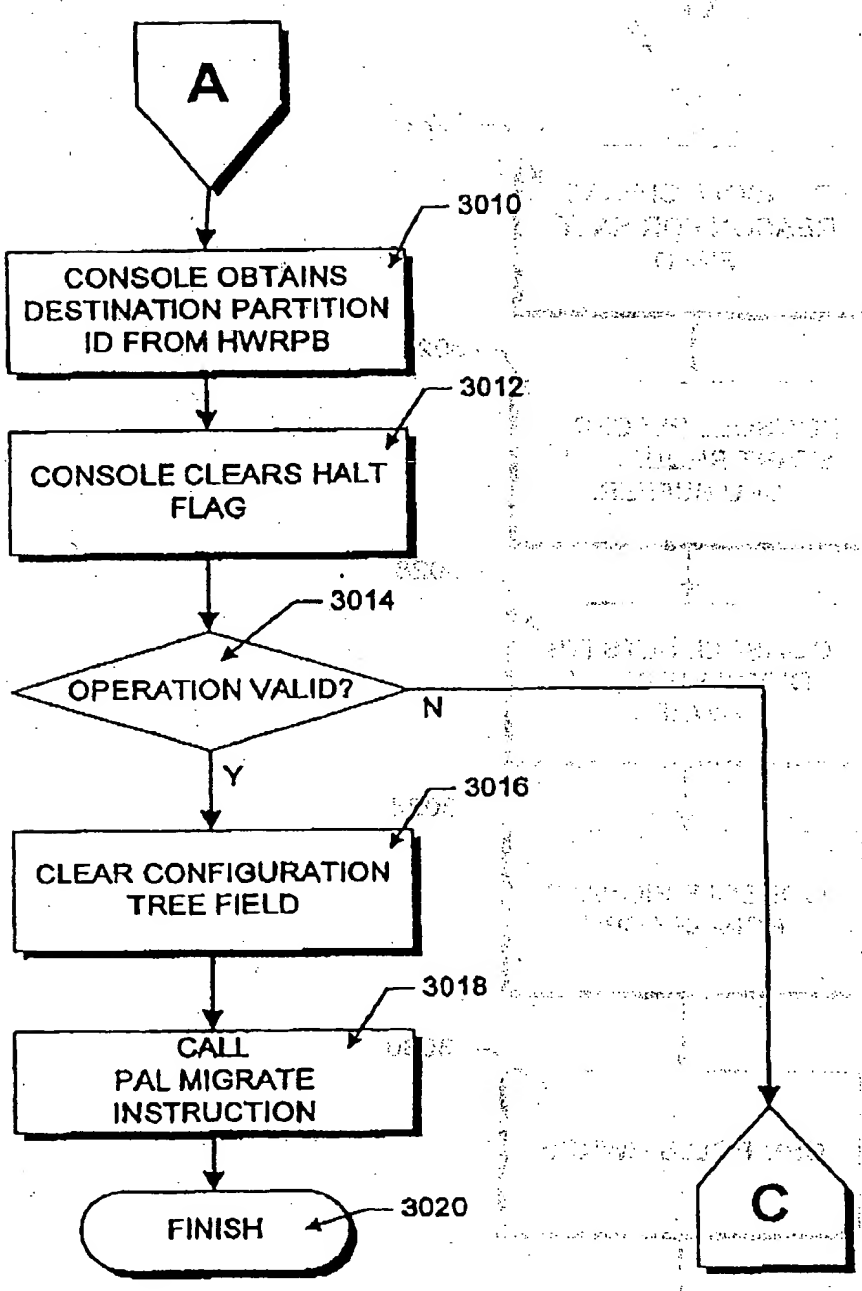

A
3010
CONSOLE OBTAINS DESTINATION PARTITION ID FROM HWRPB
3012
CONSOLE CLEARS HALT FLAG
3014
OPERATION VALID?
N
Y
3016
CLEAR CONFIGURATION TREE FIELD
3018
CALL PAL MIGRATE INSTRUCTION
FINISH
3020
C

FIG. 30B

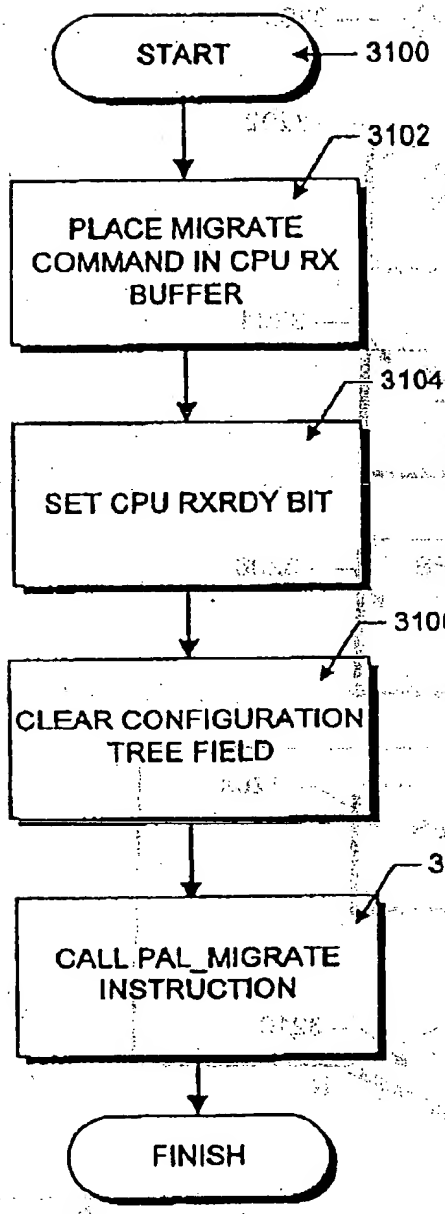

START
3100
3102
PLACE MIGRATE COMMAND IN CPU RX BUFFER
3104
SET CPU RXRDY BIT
3106
CLEAR CONFIGURATION TREE FIELD
3108
CALL PAL_MIGRATE INSTRUCTION
FINISH

FIG. 31

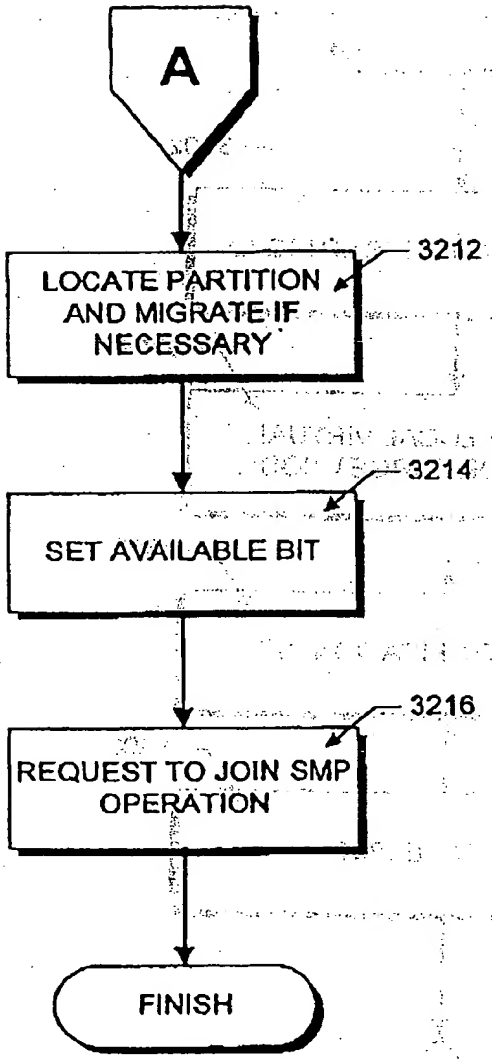

A
LOCATE PARTITION AND MIGRATE IF NECESSARY
3212
SET AVAILABLE BIT
3214
REQUEST TO JOIN SMP OPERATION
3216
FINISH

FIG. 32B

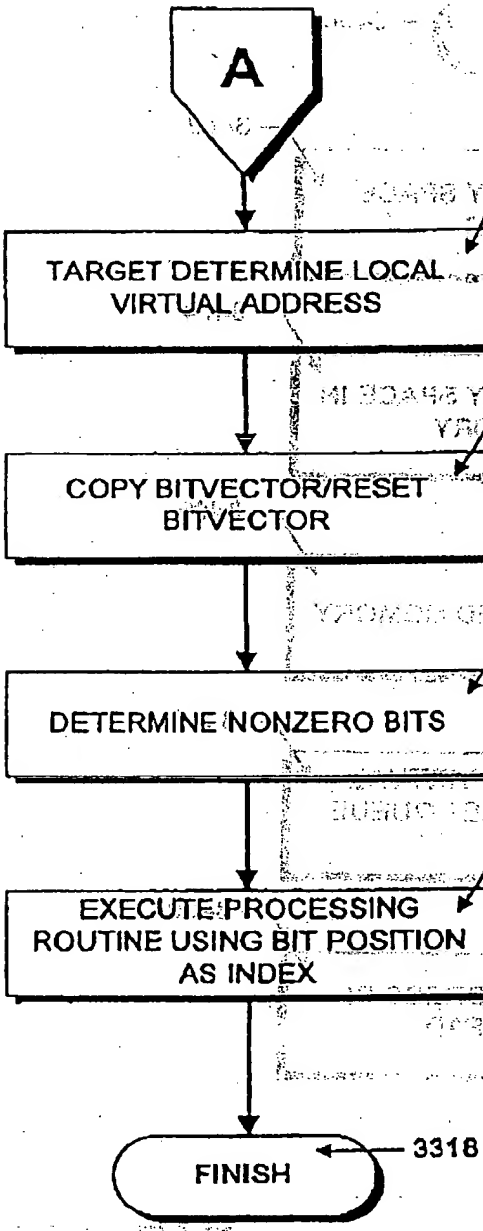

A
3310
TARGET DETERMINE LOCAL VIRTUAL ADDRESS
3312
COPY BITVECTOR/RESET BITVECTOR
3314
DETERMINE NONZERO BITS
3316
EXECUTE PROCESSING ROUTINE USING BIT POSITION AS INDEX
FINISH
3318

FIG. 33B

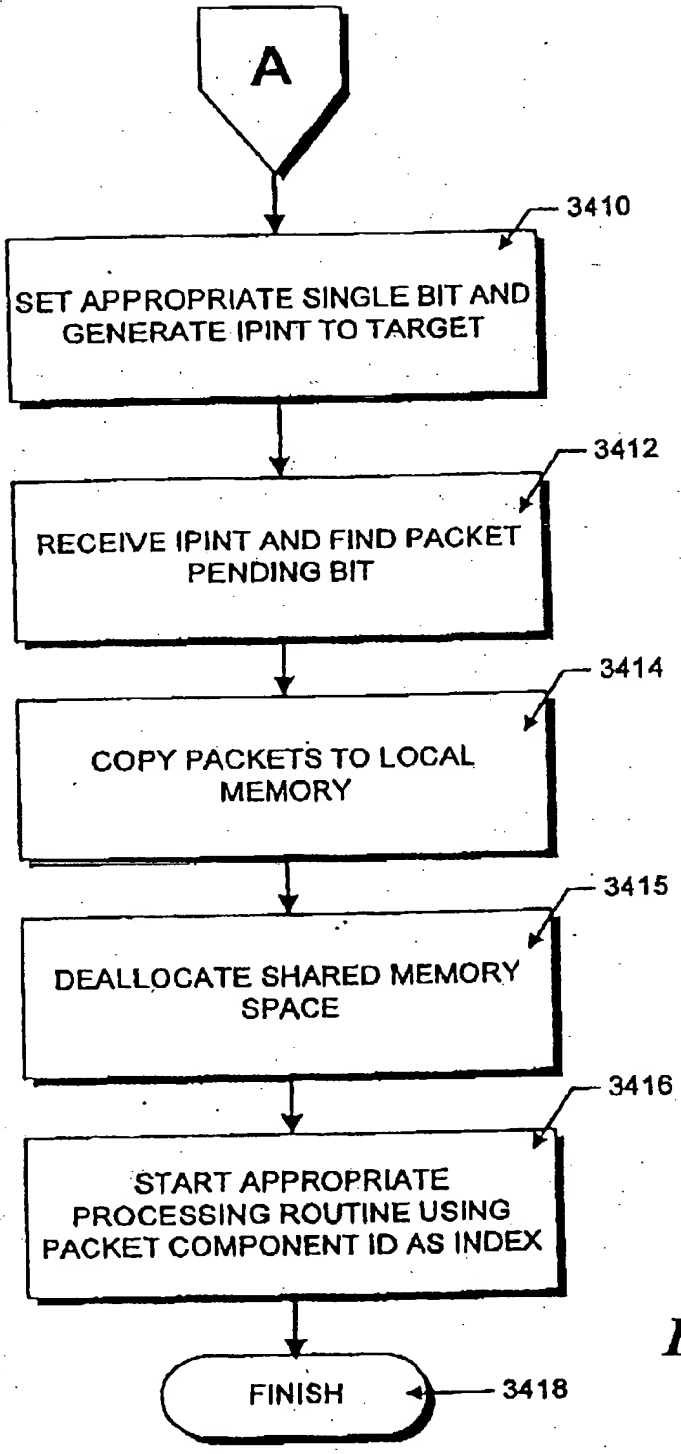

A
3410
SET APPROPRIATE SINGLE BIT AND GENERATE IPINT TO TARGET
3412
RECEIVE IPINT AND FIND PACKET PENDING BIT
3414
COPY PACKETS TO LOCAL MEMORY
3415
DEALLOCATE SHARED MEMORY SPACE
3416
START APPROPRIATE PROCESSING ROUTINE USING PACKET COMPONENT ID AS INDEX
FINISH
3418

FIG. 34B

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☐ BLACK BORDERS
- ☐ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- ☐ FADED TEXT OR DRAWING
- ☑ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- ☐ SKEWED/SLANTED IMAGES
- ☐ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- ☑ GRAY SCALE DOCUMENTS
- ☐ LINES OR MARKS ON ORIGINAL DOCUMENT
- ☑ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- ☐ OTHER: _____

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.